LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

AD-A209 117

MIT/LCS/TR-438

EFFICIENT LAZY DATA-STRUCTURES ON A DATAFLOW MACHINE

Steven K. Heller

February 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-438	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-75-C-0661
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. · PROJECT NO. · TASK NO · WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Efficient Lazy Data-Structures on a Dataflow Machine

12. PERSONAL AUTHOR(S)
Heller, Steven K.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	1989 February	108

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Dataflow, Evaluation order, Functional languages, Lazy evaluation, Parallel data-structures, Computer languages, Computer programming, Data processing. (KR)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
Eager interpreters are able to exploit vast parallelism, yet lazy interpreters have more desirable termination properties. We propose lazy data-structures, an extension to the dataflow language Id, to support a combination of eager and lazy evaluation. We describe the semantics of lazy data-structues, as well as efficient implementation on the Tagged-Token Dataflow Architecture and the Monsoon Explicit Token Store Machine. We develop support for lazy data-structures in the language, the compiler, the run-time system, the interpreter, and the proposed hardware; and demonstrate the effectiveness of the construct as well as the limitations. [illegible]

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE *(Include Area Code)* (617) 253-5894 · 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted.
All other editions are obsolete
SECURITY CLASSIFICATION OF THIS PAGE
*U.S. Government Printing Office: 1985—507-047
Unclassified

89 6 15 042

Efficient Lazy Data-Structures on a Dataflow Machine

Steven K. Heller

MIT / LCS / TR-438
February, 1989

Efficient Lazy Data-Structures on a Dataflow Machine

Steven K. Heller

Technical Report MIT / LCS / TR-438
February, 1989

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Eager interpreters are able to exploit vast parallelism, yet lazy interpreters have more desirable termination properties. We propose *lazy data-structures*, an extension to the dataflow language Id, to support a combination of eager and lazy evaluation. We describe the semantics of lazy data-structures, as well as an efficient implementation on the Tagged-Token Dataflow Architecture and the Monsoon Explicit Token Store Machine. We develop support for lazy data-structures in the language, the compiler, the run-time system, the interpreter, and the proposed hardware; and demonstrate the effectiveness of the construct as well as the limitations.

Key Words and Phrases: Dataflow, Evaluation order, Functional languages, Lazy evaluation, Parallel data-structures.

Acknowledgments

First acknowledgments go to the one-named man. Arvind has been advising me since my undergraduate days, has been a source of support, frustration, and encouragement, and has always supported my artistic (autistic?) efforts. Besides, he has a lovely family. Thanks, boss. Herr Professor Doctor Committee Member Rishiyur Sivaswami Nikhil, Goodyear Professor of High-Temperature Pneumatic Studies, Zeppelin Professor of Windbags and Wind Tunnels, was always available as a sounding board for my ideas. His diligence and humor have been inspiring. Bert Halstead, the *third member* of my committee, has had the longest lived impact on my education. He's made of "the right stuff", and I'm glad to have had an opportunity to benefit from his presence.

I was lucky enough to have two research groups. The folks from both the Computation Structures Group as well as the folks from the Parallel Processing Group were very generous in their support. Many people read drafts of the proposal and of the thesis, including: Paul Barth, Stephen Brobst, Andrew Chien, Bhaskar GuhaRoy, Bob Iannucci, Arun Iyengar, Suresh Jaganathan, R. Paul Johnson, Vinod Kathail, Jim Miller, Dan Nussbaum, Peter Nuth, Ken Steele, Ken Traub, Earl Walden, and Jonathan Young. My stay at the lab was *made* by the people with whom I had the privilege of working, especially: Andy Boughton (amazing online ass saver), David Culler (generally amazing dude), Bob Iannucci (amazing speaker — gives talks almost as well as my dad), Vinod Kathail (amazing in theory), Greg Papadopoulos (amazing name), Richard Soley (amazing stature), and Ken Traub (amazing musical coordinator), who are a group of amazing dudes, without whom, grad school itself would have been impossible. Thanks guys.

Natalie Tarbet (alias Natalie Fowler Strunk and White) edited both proposal drafts and thesis drafts (that's double plurals, folks). Her criticism was constructive and instructive. Her editing, along with the excellent editing of Debbie Howe, helping myself for translating this "works of a masterpiece" from nearing psycho-babble towards the dialectic of English fore which it now appears inside of. She also appointed herself my MIT mother, which was occasionally appreciated.

Thanks to Professors Melcher and Meyer, my academic advisors for my undergraduate career. Professor Eric Grimson, my academic advisor for most of my graduate career, has been a great source of support and advice. Dr. Hal Fleisher and the boys of 704 helped me begin my research career, and I thank them and IBM for their early support.

Professor Pingali (he used to let me call him Keshav) was my officemate for most of my graduate tenure. His example of intellectual excellence and his encouragement will never be forgotten. Besides, he has a lovely wife. To my new/old officemate Arun, I wish the best of luck. David, Sara, and Ethan are my Boston family. I don't have the words to express my appreciation. *"I don't know what to say the monkey won't do!"* Deborah Estrin, who is probably doing sit ups as she reads this, may be the only person who knows exactly how I feel right now. Besides, she has a hottub *and* a lovely family.

I have enjoyed supervising a small army of UROP students over the years, and I'd like to thank them all for participating in my management and group dynamics experiments. The *Junior Compiler Hackers*: Robert Chu (ISMC Simulator), Rich Wei (ISMC Simulator), Tim Chambers (ISMC Simulator testing), Joon Song (NETS, Parser Generator), Riaz Adamjee (NETS), Ruth Griffith (Lex Generator), Dawn Clarke (Schemata), Dinarte Morias (Id Indenter), Eric Hao (Pid Translator), Yang Meng Tan (Id Experiments), Terry Olkin (Machine Graph Enumeration), Francis Lam (Code-block Disassembly), and Chuck Fabian (Lazy Structure Hacking). Chuck also implemented some of the syntactic sugars for streams that are a part of this thesis.

I'd like to thank my many teachers, students, and friends in Tae Kwon Do from the MIT Korean Karate Club (especially Mike, Lim, and Gene), the Community Martial Arts Club (especially Bob and Chuck), and the members of BABBS. Thanks also to the folks at MIT Athletics, who supported my efforts, especially Jack Barry, Sandy Lett, and Gordon Kelly. Without Tae Kwon Do, life itself would have been impossible.

Baker House was my home for most of my graduate residency. Thanks to Professor Harold Reichie the Lord of the Manor, Irene the Housemistress, Ken Donahey the manager, Norman the Iron Man, Jong On the Psychic Girl, Leo the rugger, Stephen the Californian, and Cathy, whose caring nature brought me through several crises. Thanks also to Scotti and TJ, whose caring for others made the fourth floor. And of course, Venu, Doug, Beth, and Bob, the Midnite Madness Gang, were a pain in the abs — regularly.

Carol, Debbie, Judy, and Kalyn participated in a key part of my survival program. *I'll* give you a few clues. It can be done in pairs, groups, and even alone. It's better if you have rhythm. It's tough to do with 100% safety (S—A—F—E—T—Y). It makes you feel *real good*. Finally, without this activity — well — I'd rather not think about it.

There's a big difference between long distance and being there. Sometimes that's just too bad. Sarah d owes me a rap. Stay chill, home girl! My friend Eva, like Sidhartha, searches for the meaning of life. I hope that she finds it and herself. Your support and love are appreciated.

My family, who physically abandoned me at the start of my graduate career, was always there emotionally. Every one of my siblings managed to complete their education before I did[1]. Perhaps I'll join them in mellow-land. Special thanks to my sister Leslie, who moved from California to Boston in order to support me in my final scholastic efforts. At least that's what she told me.

Finally, I'd like to thank Arbz for no particular reason, and I'd like to thank Toes for no reason at all.

I've been at MIT for a long time, and I've had the good fortune to have many friends and supporters, and I could go on and on and on. I wish that I could ... . For all you dudes (and dudettes), this Ph.D.'s for you!

In a small grey box[1], I found the following discussion attributed to Darold Treffert, a psychiatrist who has studied savants:

> ... *Autistics seem to have trouble sorting out sensory stimuli, experiencing the* world "like a television set with all the channels coming in at once," as Treffert puts it. Their rituals and obsessions may be an attempt to impose order on this jumbled and static-ridden world. For a few, a monomaniacal focus on calculating prime numbers, say, may have the same effect.

$$\begin{aligned} P(300) &= 1987 \\ P(P(134)) &= 5749 \\ P(2,400,000) &= 39,410,867 \\ P?(02143) &= :True \end{aligned}$$

-bf

This research was supported by Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0661.

[1]*Q: How many MIT grad students does it take to change a lightbulb? A: Only one, but it takes nine years!*

I think that we all share a common experience. When we are young, our parents tell us, "Thank God we're healthy!" Children cannot understand this nor can they appreciate it.

To my friends BG, Edgar, Josef, and Sarah

—

You have championed over difficult times
and are all on your way to renewed health.
Thank God you're healthy!
Sending you my most positive energies.

and
To the memory of Mrs. Grecco

Contents

Chapter 1

Introduction

Functional languages support a powerful programming methodology based on higher-order functions and infinite objects. Furthermore, they admit diverse implementation strategies. While lazy functional languages have more expressive power than eager languages, eager functional languages are more amenable to parallel execution. The goal of this thesis is to extend the power of the dataflow language Id [37], a non-strict eager language, and its implementations [7, 38] by introducing a limited form of lazy evaluation; lazy data-structures allow the evaluation of the contents of a data structure slot to be postponed (perhaps forever) until a consumer reads the slot, thus extending the expressive power of Id.

Several terms require clarification. Section 1.1 discusses what we mean by the terms "strict", "non-strict", "lazy", and "eager" as ways of describing functional languages and interpreters. Section 1.2.1 presents two interesting programming paradigms that are facilitated by lazy data-structures, and Section 1.2.2 discusses the costs of lazy evaluation.

1.1 Terminology

Most programming languages are strict (*i.e.*, have strict semantics). This means that the arguments to a procedure are evaluated before the procedure is called. Consider the following expression that constructs a pair using the **cons** procedure and selects the first element of the pair using the **head** procedure:

```
head (cons exp1 exp2)
```

Since the argument expressions **exp1** and **exp2** are evaluated before calling **cons**, **exp2** is evaluated even though the result of the evaluation is not needed to produce the overall result.

If the evaluation of either **exp1** or **exp2** diverges (goes into an infinite loop), evaluation of the overall expression diverges too, and the overall expression never produces a result. Termination and producing a result for the entire expression depend on the termination of **exp2**, even though the result is not needed.

One exception to the "arguments first" rule is allowed. The consequent and alternate of a conditional expression, which can be syntactically identi User not logged in or not receiving messages. 1· the predicate is resolved only one of them is evaluated. This is necessary to permit recursive definitions. It is not possible for users to define conditionals with procedures, however, as we discuss in Section 1.2.4.

Lazy functional languages are a subset of non-strict functional languages. Computation is never performed unless the result is required to produce the overall result. In the example above, in a lazy functional language, the overall result does not depend on **exp2**, and **exp2** would not be evaluated. This is accomplished by calling procedures before evaluating arguments: the evaluation of an argument is delayed until its value is found to be required to produce the result. A technique for delaying the evaluation of an expression is described in Section 1.2.2.

Lazy functional languages have an inherently sequential aspect, too: procedures are called before arguments are evaluated. The compiler may compile code to evaluate an argument before or in parallel with calling a procedure, for example, as long as it can be sure that the semantics of the program are the same. *Strictness analysis*, a technique for accomplishing this type of optimization, is discussed in Section 1.2.3.

Since an expression is evaluated only if its result is required to produce the overall result, an unbounded amount of computation is never performed once the answer has been computed. Said another way, producing a result and termination are inextricably tied. If a computation does not terminate, no result will be produced.

Lazy functional languages do not cover all non-strict functional languages. Id [1] [37] is a non-strict non-lazy language. Producing the result of an Id program does not imply termination. In the above example, in Id, production of a result depends on termination of the evaluation of **exp1**, and overall termination depends on the termination of both **exp1** and **exp2**. "Non-lazy" is called "eager", the complement of "lazy". An expression might be evaluated even if its result is not a prerequisite for the overall result.

[1] Id is not a functional language, but a large subset of Id is functional. When the non-functionality is an issue, it will be pointed out.

Just as non-strict functional languages subsume lazy functional languages, eager functional languages subsume strict functional languages as seen in Figure 1.1.

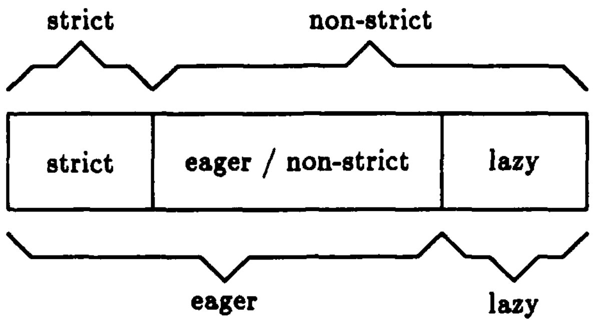


Figure 1.1: The Functional Language Spectrum

So far, we have talked about attributes of languages. These qualifiers can also be applied to interpreters, machines on which programs can be run. An eager interpreter naturally supports the interpretation of an eager language, *etc.* That is not to say that a lazy functional language cannot be implemented on an eager interpreter, as we will see in several examples.

In the rest of this chapter, we discuss the motivation for our lazy/eager mixture, and we discuss other dataflow approaches for achieving laziness.

1.2 Lazy Evaluation Reconsidered

In this section, we consider various aspects of lazy evaluation. We begin with some programming paradigms available in lazy functional languages but not eager functional languages. Then, we consider the cost of lazy evaluation and how that cost is avoided (to some extent) in lazy functional languages. Finally, we discuss why we chose an eager language rather than a lazy language as a starting point, and we present our approach.

1.2.1 Two Interesting Programming Paradigms

Lazy functional languages facilitate programming paradigms not available in eager languages. Two such paradigms are considered in this section: programming with conceptually infinite data-structures and programming with data structures with slots that are expensive to compute.

Streams are classic examples of infinite data structures. These conceptually infinite lists expand only as far as their consumers require. As long as consumers only traverse a finite prefix of a stream, only a finite prefix is computed. More generally, very large or infinite data structures can be traversed while newly explored sections are generated automatically as needed.

Sometimes particular data structure positions are expensive to compute, but we might ignore the value of the position. Consider, for example, a memoization table for a complex function. For every position that we avoid computing, significant savings are realized. We call this paradigm "programming with expensive slots".

Both of these paradigms, which are closely tied to data-structures, are made available by the system developed in this thesis.

1.2.2 The Cost of Lazy Evaluation

When an expression is evaluated, the interpreter (machine) has some state associated with it. In particular, the free variables of the expression are available. In order for the evaluation of the expression to be delayed, provision must be made to make the values of the free variables available when the expression is eventually evaluated. One technique for packaging a delayed expression with its environment is called thunks[2]. A thunk is a piece of code to evaluate the delayed expression after "restoring the expression's environment".

Many efficiency issues arise when thunks are used to implement delayed computation. First we consider the general issues of sharing the result of a delayed expression and of having more thunks than necessary to achieve desired program behavior.

If two computations have *independent copies* of a thunk and both computations evaluate the thunk, the computation will be performed twice. Sharing computation is so important that any realistic approach must incorporate it. Any implementation which does so must provide storage for the result of each thunk. The need for a shared location motivated our restriction (to be presented) that delayed expressions sit in data structures.

Coordination of a shared resource requires synchronization; in a sequential machine, where only one process is active at a time, the synchronization and coordination that surrounds a shared thunk are straightforward. In a parallel machine, however, managing any shared resource is complex. The embodiment of the required synchronization mechanism will emerge as the basis for our approach.

Most expressions are always computed and need not be delayed. Strictness analysis and

[2]The term *thunk* comes from Algol60, where it was the mechanism used to pass call-by-name parameters. Although the value was not computed yet, the compiler had already "thunk" how it would be done when the time came [18]. Each time the variable was needed, the value was computed.

similar techniques optimize the implementation of lazy functional languages by eliminating as many superfluous thunks as can be identified. Since the starting point is far from the target, these techniques must be excellent to provide reasonable efficiency. We shoot for the target from close range. Annotations point to the few lazy exceptions.

The cost of thunks can be broken up into three categories: the cost of building thunks, the cost of the first reference, and the cost of successive references.

When an expression is delayed, the environment must be saved. In a system where environments are manipulable (such as Scheme [41]), the thunk for the delayed expression can simply record a pointer to the delayed expression's lexically enclosing environment. This environment, however, is likely to contain much more information than we need to evaluate the expression. Since the shared environment already exists, using it makes the construction of the thunk fast, but the lifetime of the entire environment is now tied to the thunk. The environment (which is actually a sequence of frames in Scheme) cannot be reclaimed until the delayed expression is evaluated or discarded. This is known as *dragging*: the thunk is dragging the environment. Another alternative is to copy the subset of the environment corresponding to the free variables of the delayed expression into an independent environment. Building a new environment takes more time, but dragging is avoided.

The first time the value of a delayed expression is requested, the expression must be evaluated. Assuming that we do not wish to recompute the expression, we must keep track of both the value of the expression the fact that the expression has been evaluated. All references, both initial and successive, must check a flag to determine if the delayed expression has been evaluated. We would like to minimize the cost of this recurring expense.

1.2.3 Decreasing the Cost of Lazy Evaluation in Lazy Functional Languages

The lazy functional languages community has developed techniques to abate the cost of lazy evaluation. Strictness analysis and strictness annotations allow some expressions to be evaluated eagerly. Path analysis [48] uses several different kinds of thunks, allowing various special cases to be optimized.

Strictness Analysis and Strictness Annotations

Every time an expression is delayed, there is a cost. Strictness analysis is an automatic technique and strictness annotations is a programmer controlled technique to decrease the cost of lazy evaluation by computing some expressions eagerly.

Most functional languages have lazy semantics, and the compiler is expected to determine which expressions can be evaluated eagerly. Progress has been made in the area of strictness of higher order functions [25] and non-flat domains [20], but the problem is far from solved.

A function is said to be strict in an argument if the value of that argument is required for the function to produce a value. Notationally, the function `f` is strict in its second argument if for all `x`, $\texttt{f}(\texttt{x},\bot) = \bot$, where $\bot$ denotes an undefined or non-terminating computation.

Hudak and Young show that the problem of first-order strictness analysis is complete in exponential time and attribute the result to Meyer [25]. They go on to explain that, since "the size of most functions is small, the complexity seems to be tractable in practice."

It is worth noting that any programmatic strictness analysis technique must be an approximation. Determining the strictness of a function asks a question about the termination behavior of the function and is clearly undecidable, in general. In their section on the correctness of their algorithm, Hudak and Young explain that their technique is *safe* as it never declares a function strict that is not. So, even approximate techniques are complex. We give an example demonstrating the difficulty of strictness analysis in Section 5.2.

To assist the compiler, annotations are sometimes provided to the programmer to declare strictness properties that the compiler would otherwise have to deduce. In the remainder of this section, we will discuss the use of strictness annotations in Miranda[3] [44] and FLIC [39].

Miranda allows the programmer to annotate algebraic type constructor definitions to be strict in particular arguments. In the following type specification, streams of numbers are defined to have strict heads and lazy tails. The head is strict by virtue of the `!` annotation, and the tail is lazy by default. `scons` stands for stream cons.

```
stream ::= empty | scons num! stream
                                                    Miranda
```

The strict/lazy argument pattern establishes a calling convention for each data-structure constructor. The caller is compiled to pass certain arguments as values and the rest as delayed expressions, and the constructor is compiled to receive that pattern. No provision is available, however, to annotate general procedure definitions as having strict arguments. This is probably because strictness analysis for procedures was better understood at the time that Miranda was

[3]Miranda is a trademark of Research Software Ltd.

designed than was strictness analysis for data structures, a more recent development. However, it is not possible to annotate built-in data type constructors such as **cons**, the list constructor. Although the user can define new algebraic types with any strictness pattern desired using the built-in types, the special syntax for supporting lists is lost, including *dotdot notation* for constructing arithmetic sequences and recurrences; and *list comprehension* for generating, mapping, and filtering lists. Dotdot notation and list comprehension, based on Zermelo-Frankel set notation, are expressive ways to specify lists. Furthermore, Miranda provides no facility to annotate actual parameter expressions as strict. We will return to this prospect presently.

Warren Burton proposes a variation of Miranda [15]. Procedures are called with strict semantics, and data structure are constructed with lazy semantics. "Partially strict pseudo-constructors" are procedures with annotations attached to their type specifications. The annotations indicate varying degrees of laziness. The following code defines the constructor for a stream of numbers with strict heads and lazy tails. The head is strict by default (procedures have strict arguments by default), and the tail is lazy by virtue of the **name** annotation.

```
scons :: * -> name [*] -> [*]
scons a b = a:b
```

Warren Burton's Miranda

Since the annotations are attached to procedure definitions, which are strict by default, the annotations should probably be called laziness annotations. Three modes are established for passing procedure arguments. Call by **value** is the default and requires no explicit annotation. Call by **speculation** is an eager variation for parallel machines, where the argument and procedure can be evaluated in parallel. Call by **name** is a lazy evaluation technique, but values are not memoized. Burton argues that it may be cheaper for a different processor to recompute a value rather than to send the computed value from one processor to another, especially in the case of a data structure. Besides, the programmer can provide for sharing explicitly.

FLIC (Functional Language Intermediate Code), as its name indicates, is not intended to be a front-end language. As FLIC is intended to support any number of functional languages targeted at any number of sequential or parallel architectures, complete facilities are available for indicating strictness[4]. A procedure can be marked as strict in its argument (FLIC procedures

[4]FLIC defines an annotation to contain purely pragmatic information which can be deleted to derive the semantics (meaning) of a program. We will take a looser interpretation of the term to include proper programming constructs.

have only one argument), covering both general procedures and data structure constructors. Closely related (and equivalent) is a sequentialization function that takes two expressions and returns the second, but not until the first completes. The STRICT and SEQ primitives are defined by the following equations:

```
STRICT f ⊥ = ⊥
STRICT f x = f x

SEQ ⊥ b = ⊥
SEQ a b = b
                                                        FLIC
```

The following code[5] defines **scons** as a stream constructor that has strict heads and lazy tails. The **strict** function ensures that all first actual parameter expressions of **scons** are reduced to values before **scons** is applied.

```
scons = STRICT (\x\y cons x y)
                                                        FLIC
```

We will consider sequences of reductions that demonstrate the use of the **scons** and **strict** functions. *head* and *tail* are stream selectors.

```
head (x:xs) = x
tail (x:xs) = xs
                                                        FLIC
```

In the following sequences, expressions that are underlined are about to be rewritten.

```
head (scons 5 ⊥)
⇒ head (STRICT (\x\y cons x y) 5 ⊥)
⇒ head ((\x\y cons x y) 5 ⊥)
⇒ head (cons 5 ⊥)
⇒ 5
                                                        FLIC
```

```
tail (scons ⊥ 5)
⇒ tail (STRICT (\x\y cons x y) ⊥ 5)
⇒ tail (⊥ 5)
⇒ tail ⊥
⇒ ⊥
                                                        FLIC
```

[5]Backslash is FLIC's symbol for lambda.

As a more complex example consider the definition of **strict_cons**, the pair constructor that is strict in both arguments:

```
strict_cons = STRICT (\x (STRICT (\y cons x y)))
                                                    FLIC
```

FLIC also provides facilities for evaluating an expression before applying a procedure. The following let expression binds the value of the argument expression **arg_exp** to the name **arg_val** and applies the procedure **foo** only after **arg_val** reduces to a value.

```
= arg_val arg_exp (SEQ arg_val (foo arg_val))
                                                    FLIC
```

FLIC also provides annotations (as opposed to the preceding, which were proper language constructs) to indicate both formal and actual argument strictness. These annotations are to be used by the compiler after performing strictness analysis, for example.

FLIC takes a general approach. Not only can a particular procedure be established with a mixed strict/lazy calling convention, but actual parameter expressions can be marked independently, providing additional opportunities for savings. In conventional implementations, the various mixed calling conventions are necessary as values and delayed expressions cannot be freely interchanged. A function has to know what was potentially delayed, and what was a value. Even so, an actual parameter in a position that is normally passed as delayed can be marked as strict to some advantage. The delayed structure (that contains a flag and the delayed expression or the value of the expression) can be marked "evaluated" and the value can be computed directly and stored. A delayed structure still has to be allocated to satisfy the calling convention, but the expression need not be delayed, and can even be computed inline, providing additional savings. In this way we have a hierarchy of mechanisms corresponding to increasing efficiency.

1. lazy: all arguments are passed unevaluated
2. definitions are annotated: a procedure call is established for each strict/lazy combination, and some arguments are passed evaluated, some, unevaluated
3. applications are annotated: more arguments may be strict but still packaged as evaluated delayed-expressions

4. strict: all arguments are evaluated

Bloss, Hudak, and Young [14] develop an infrastructure which can take advantage of these kinds of situations which we discuss briefly in the next section.

In a lazy functional language, it is particularly effective to annotate definitions as most expressions can be eagerly evaluated and one definitional annotation covers an entire class of instances. In an eager language, however, we do not wish to proliferate laziness casually, as it is rarely needed. For this reason, we use annotations only in actual expressions.

If values and unevaluated expressions can be freely interchanged (which implies implicit forcing), as they can be in Multi-Lisp [21], annotations at the definition and at the application are equivalent, the former being an abbreviation for many of the latter. As a result of the interchangeability, no special calling conventions are necessary. Such a design decision, however, requires architectural support for an efficient implementation, lest we require repeated explicit checks. In an implementation on stock hardware with no architectural support, such as Multi-Lisp on Concert [22] or Mul-T [30] (a dialect of Multi-Lisp and T compiled for the Encore Multimax), this turns out to be quite expensive.

When designing hardware, however, providing support for trapping delayed expressions is usually an easy extension. Consider Lisp machines, for example, which trap to microcode on all sorts of exceptional cases, or SOAR [45] (Smalltalk on a RISC) or SPUR [19] which trap to software in exceptional cases. The idea is to handle common cases quickly while trapping and paying a penalty in the less frequent exceptional cases.

We restrict our attention to data structures and develop a mechanism that is transparent to the consumer and depends on hardware support for an efficient implementation. Annotations will always be always associated with the actual construction of data structures.

Path Analysis

Hudak *et al.* develop a technique called *path analysis* for optimizing implementations of lazy functional languages [14]. The compiler [48] tries to determine that a particular use of a thunk is the first or the last, or that it cannot be the first (*i.e.*, the delayed expression is already evaluated), by tracing the path through the sequential execution. Each special case has opportunities for optimization.

Four ways of forming thunks for procedure arguments are described. The list includes the standard Henderson-style self modifying procedure, two forms of flag and procedure/value cells, and, for completeness, a non-delayed mechanism. The four methods, or *modes* are:

1. Closure Mode: A delayed expression is wrapped in a procedure. The procedure may or may not cache the value when it gets evaluated. In this mode, the compiler can rewrite `(DELAY (FORCE x))` as `x`, for example. Two disadvantages are pointed out. Each use of the thunk requires a general procedure call, which is expensive. Context specific optimizations such as order of evaluation with respect to the caller are not available.

2. Cell Mode: A thunk is represented as a pair, a flag and a closure or value, depending on the flag. The closure mode problems vanish, but some optimizations are precluded due to their interaction with even other optimizations[6].

3. Optimized Cell Mode: The delayed function's arguments are guaranteed to be unevaluated on entry. While `(DELAY (FORCE x))` can no longer be rewritten as `x`, other optimizations are enabled. The assertion about the function's arguments increase greatly the opportunities for using path analysis.

4. Value Mode: The value is computed directly. This mechanism is included for completeness.

Ordering in a parallel system, however, is much less restrictive, and opportunities for such optimizations are significantly diminished. In a dataflow system, for example, the program captures only a partial order of the operations.

1.2.4 Why Not Laziness

Some proponents of lazy functional languages argue that lazy functional languages support equational reasoning, but strict functional languages do not. Equational reasoning allows function definitions to be treated as equations or identities in the sense that a compiler can substitute them without changing the meaning of programs. Consider the following definition of the pair selector **head**. We would like to be able to view it as an equation also, relating **head** with the pair constructor **cons**.

```
    head (cons x y) = x
```

[6]An example demonstrating this point would take a great deal of development. The interested reader should consult [14].

In a lazy functional language, this equation holds even if the evaluation of y diverges; since y is not needed, it would never be evaluated. Whenever the compiler sees the expression `head (cons x y)` in a program, it can substitute y without changing the meaning of the program. In a strict functional language, however, both x and y are evaluated before cons or head are applied. If the evaluation of y diverges, we cannot proceed, even though the value of y will be discarded. If the compiler substitutes y for `head (cons x y)`, it might change the termination behavior of the program.

All non-strict functional languages, not just lazy languages, support equational reasoning [11]. In an eager non-strict functional language, the argument expressions can be evaluated in parallel with each other and in parallel with procedure application. Furthermore, the notions of "getting an answer" and "terminating" are separated, and, as a result, the equation makes sense. The interested reader is referred to [33] for additional details.

Some proponents of lazy evaluation claim that lazy evaluators are more efficient than eager evaluators [11] because they perform the minimum number of reduction steps to find normal form. But more interpretation is required to decide which reduction is next, and to decide if a particular expression has already been evaluated. A converse claim is of interest: if a lazy interpreter and an eager interpreter take the same number of reduction steps to reach an answer (normal form), then the lazy interpreter did at least as much total work as the eager interpreter. Work includes both reduction and interpretation.

A lazy evaluator is necessarily more complex and therefore more expensive than an eager evaluator, as pointed out in [13]. This expense is reduced by strictness analysis [25], and other similar techniques [14]. Most programs need none of that power, however. Even programs requiring lazy evaluation need it for only a small fraction of the program. This assertion will be demonstrated by overwhelming evidence.

Control Structures Versus Data Structures

Lazy evaluation might leave expressions unevaluated in two ways — both arguments to procedures and data structure slots may be left unevaluated. The computation of an actual argument to a procedure may diverge, yet the value of that argument may not be needed. This is the situation if we treat conditionals as procedures and consider recursive definitions. Consider the following Id definitions[7]. **typeof** declares the type of an identifier. No type declarations are

[7]All code in this document is written in Id unless otherwise marked.

required by Id; they are included for documentation purposes.

```
typeof my_if = B -> *0 -> *0 -> *0;
def my_if p c a = if p then c else a;

typeof fact = N -> N;
def fact n = if n<2 then 1
             else n * fact (n-1);
```

If we replace `if` by `my_if` in the preceding definition of `fact`, calls to `fact` will not terminate.

In the case of data structures, an "infinite object" can never be fully expanded, yet these objects are sometimes convenient for programming. With the notable exception of the conditional construct, we hypothesize that the lazy evaluation of control structures is rarely needed, and the lazy evaluation of data structures is needed infrequently. The utility of our approach, which disallows the former and facilitates the latter, can only be measured by its practical effectiveness. Chapter 4 examines the strengths and weaknesses of this choice.

In concentrating our effort on data structures, we are not alone. Miranda's strictness annotations apply only to data structures. In Warren Burton's variation of Miranda where procedures are called strictly and data structures are constructed lazily to varying degrees, based on annotations. Conversely, Multilisp's futures are oriented around expressions.

Explicit Allocation of Storage

While the producer and consumers of the value of an expression have no storage automatically and naturally associated with them, the producers and consumers of a data structure meet at the data structure itself. The data structure provides a meeting place, and the data structure operations provide a point in time for orchestrating the delaying and forcing of expressions.

Sequentialization and Parallelism

The main source of parallelism in functional languages is the ability to evaluate all arguments to all procedures in parallel. Strict functional languages require barrier synchronization to insure that all arguments are computed before a procedure is called. Similarly, lazy functional languages apply procedures before evaluating arguments, a different form of sequentialization. In both of these cases, a compiler can relax ordering restrictions if it can prove that the semantics of the program are the same. Both ends of the spectrum, however, are inherantly sequential, and sequentialization comes at the price of parallelism.

1.2.5 Our Approach

On one extreme is dataflow, an eager approach amenable to parallel execution. On the other extreme is the more powerful lazy approach. We examine an intermediate approach on the eager/lazy spectrum that offers most of the power of lazy evaluation, and the efficient parallel implementation that comes with dataflow.

If we only delay expressions explicitly associated with data structures, an interesting compromise is achieved. The TTDA, the first architectural model for dynamic dataflow, already synchronizes array producers and consumers in hardware using I-structure memory [9, 10, 23]. A similar synchronization mechanism is required to support delayed expressions that sit in data structure slots. By generalizing I-structures, we can support both demand propagation and producer/consumer synchronization in hardware.

We develop lazy data-structures for the dataflow language Id. An expression destined for a lazy data-structure slot remains unevaluated until the slot is read, *i.e.*, until the value of the expression is requested. Lazy structures in an otherwise eager system thus provide a combination of eager and lazy evaluation. Lazy data-structures admit programming with infinite data-structures and programming with data structures with expensive slots.

The language we develop in this thesis, which we name Id#, is eager and non-strict. The evaluation of certain expressions which are always associated with data structures can be delayed.

The language, compiler, and run-time system extensions discussed in this thesis have been implemented[8]. Our graph interpreter [33] has been extended as well, providing an opportunity for experimentation. These extensions are currently being installed on our first hardware prototype processor [38].

1.3 Force and Delay

In this section, we discuss the applicability of Henderson's **Force** and **Delay** model, a language level solution that is the basis for many lazy interpreters. **Force** and **delay** are sufficient to implement a lazy functional language over an eager interpreter [24], and Id can accommodate these with higher-order procedures. However, there is no natural way to provide "memoization" within a functional framework.

[8]There are a small number of exceptions.

Procedural abstraction cannot be used to define **delay** in an eager interpreter, as the argument would be evaluated before it were passed to the **delay** "procedure". "=>" represents a macro-style source-level program transformation. A macro or special-form would be used in Scheme to achieve the desired behavior. Consider the following Scheme definitions of **delay** and **force** which, in Hudak's terminology, stores thunks in *cell mode*:

```
; delay and force in Scheme
(delay <exp>) => (cons 'delayed (lambda () <exp>))

(define (force delayed-exp)
  (if (eq 'evaluated (car delayed-exp))        ; test flag
    (cdr delayed-exp)                          ; get memoized value
    (let* ((delay-function (cdr delayed-exp))  ; get thunk
           (evaled-exp     (delay-function)))  ; evaluate thunk
      (set-cdr! delayed-exp evaled-exp)        ; memoize result
      (set-car! delayed-exp 'evaluated)        ; set flag
      evaled-exp)))
                                                              Scheme
```

A thunk is stored in a cons cell. The **car** indicates whether or not an expression has been evaluated. The expression is captured in an unapplied function and stored in the **cdr** part. The first time a delayed expression is forced, the value is remembered, and the flag is changed.

There are several problems in implementing such a scheme on a parallel machine as synchronization is required: the **force** procedure must manipulate the delayed object atomically, and the flag and data change. There are no facilities to support this behavior in Id -- the model must be extended to allow the desired behavior.

We can introduce a semaphore and the ability to change the flag and data. The semaphore would be acquired before manipulation of the delayed expression began. Unfortunately, this requires extra time to manipulate the semaphore as well as space to maintain it. The time overhead could be eliminated by using the flag for synchronization as well as indicating the evaluation status of the expression. This is possible if we allocate space for both the delayed and the evaluated expressions, as expressed in Id below. **Exchange** is an atomic operation that places a value in a structure slot and returns the previously stored value. **evaluation_flag** is a new enumerated type. Consider the following extended Id definitions of **delay** and **force**. The bodies of both **delay** and **force** are *let expressions*, consisting of bindings and a result expression (after the **in**). All variables bound in a let expression are lexically scoped, as are all variables in Id.

```
% force and delay using the flag for synchronization
% We use => since there are no macros or special-forms in Id

type evaluation_flag = delayed | evaluated;

delay <exp> =>
  {delayed_exp    = I_array (0,2);
   delayed_exp[0] = delayed;     % flag stored
   def delay trigger  = <exp>;  % thunk created
   delayed_exp[1] = delay        % thunk stored
   in
     delayed_exp};


def force delayed_exp =
  {flag = exchange delayed_exp 0 evaluated  % set and test flag
   in
     if flag == evaluated then
       delayed_exp[2]                        % get memoized value
     else
       {delay_function = delayed_exp[1];    % get thunk
        evaled_exp = delay_function 0;      % evaluate thunk
        delayed_exp[2] = evaled_exp         % memoize result
        in
          evaled_exp}};
```

This solution is very close to the standard one, and, as such, has the standard inefficiencies. We have not taken advantage of our ability to influence the architecture. A similar synchronization problem has already been solved by I-structure memory in synchronizing producers and consumers of data structures. Since a small number of states are required to capture the above behavior, we can solve our new problem efficiently by augmenting the hardware.

We do not adopt a source-to-source Henderson-style system. The framework we develop, however, is powerful enough to embed such a system, and we will discuss this embedding in Section 5.3.2.

1.4 Background: Id and Dataflow

We assume the reader is familiar with functional languages and graph reduction. In the following sections, we give a brief introduction to the language Id and dataflow computing.

1.4.1 The Id Language

Id was developed at the University of California at Irvine [5] and has evolved through several revisions at MIT's Laboratory for Computer Science. The research in this thesis was coincident with the development of the latest version [34] and, as a result, some of the "new ideas" presented herein are already in the current language document.

Id is an eager non-strict declarative language that supports higher-order procedures. Id's single assignment syntax will be introduced as we proceed.

Id programs are compiled into dataflow graphs which capture the data dependences of the program [43]; nodes correspond to operations, and arcs correspond to data dependences. The dataflow graphs can be executed directly on dataflow machines.

1.4.2 Dataflow Machines

Dataflow machines provide a vehicle for the execution of dataflow programs. Parallel architectures designed to provide cheap synchronization and tolerate long memory latencies, TTDA [7] and Monsoon, an Explicit Token Store machine [38], are *tagged token* dataflow machines as they support general purpose computation.

Values travel along arcs of the dataflow graph as *tokens*, and the machine enables operation nodes for execution by detecting the arrival of a matching pair of tokens.

TTDA and Monsoon support data structures with I-structure memory [9] which provides, in addition to the usual memory operations, operations that are useful for parallel computing. For example, producers and consumers can be synchronized on a per element basis: if a consumer arrives before a producer, the consumer is delayed automatically until a value arrives.

1.5 Other Dataflow Approaches

We discuss two other approaches, Pingali's and Amamiya's, for achieving laziness in a dataflow environment.

1.5.1 Pingali's Demand-Driven Interpreter

Pingali proposes a source-to-source program transformation for achieving lazy behavior within an eager interpreter [40]. Although we take a different direction, Pingali's work provided both the semantic base and the motivation for the work in this thesis.

Pingali's approach offers the power of a lazy interpreter within the framework of dataflow, but it is difficult to implement in practice. We briefly describe Pingali's approach and the problems that arise from it.

Each program graph is overlaid by a complementary demand graph that propagates tokens representing demands explicitly. A fork in a dataflow graph duplicates a token so that it may be sent to more than one consumer. In Figure 1.2, which depicts a transformed fork (a fork along with its demand graph), the program graph "points down" and the demand graph "points up". The bow-tie shaped nodes pass along the data tokens (the ones going down) when both the data and demand tokens arrive. The **d-union** node is a consuming merge: it forwards the first token it receives and discards the rest.

Suppose the result of a subexpression is shared, but not all consumers demand the value. The fork that distributes the value is left with residual tokens — one for each inactive fork arm.

y1 = y2 = x input: x outputs: y1, y2 input-demand: x-d output-demands: y1-d, y2-d

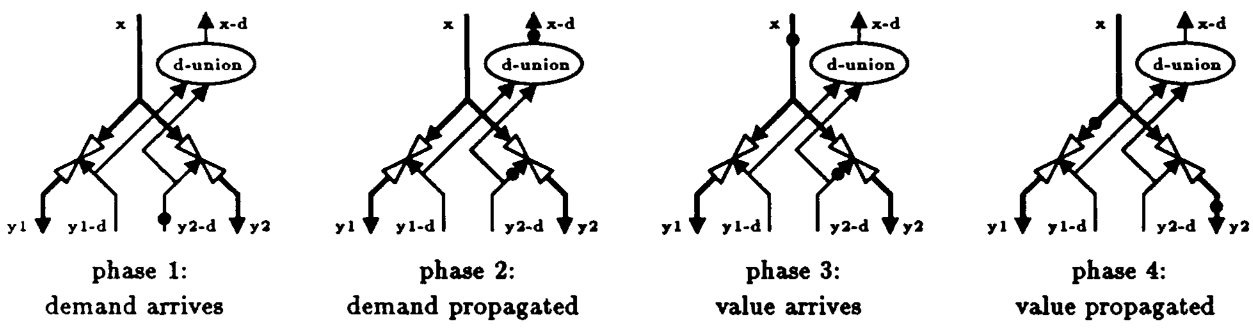


Figure 1.2: Dynamic Behavior of a Demand-Driven Fork

Figure 1.2 illustrates the fork problem. In Phase 1, a demand arrives for **y2** on **y2-d**. The demand token is duplicated; one copy waits at the gate on the right, and the other propagates through the d-union to **x-d** (Phase 2). Eventually **x** arrives (Phase 3). The value token is duplicated; one copy meets its partner at the right hand gate and passes on as **y2**, and the other waits at the left hand gate, in case **y1-d** ever arrives (Phase 4). If **y1** is never demanded, a token remains in the graph. A residual token in a dataflow graph acts like a pointer to the enclosing procedure invocation frame and prevents it from being reclaimed.

The repercussions of the unclaimed resources are more severe than they may seem at first. The delaying environment must be maintained until the delayed expression is evaluated. Even if it is evaluated at some point, the lifetime of the delayed expression may be independent of the lifetime of the environment in which it was generated, in which case, the delayed expression will drag the delaying environment.

This lifetime coupling problem cannot be ignored: if values are always demanded by all possible consumers, lazy evaluation saves us nothing. Similarly, infinite streams always have unevaluated tails. By taking a restricted view of lazy evaluation, we can deal with these issues.

1.5.2 Amamiya's Approach

Amamiya implements lazy data-structures by putting gates at the inputs of the subgraphs to be delayed [2]. When the slots are demanded, a token is sent back into the graph, allowing the computation to proceed. Amamiya's graphs also suspend indefinitely when a delayed slot is never demanded.

Amamiya also describes a mechanism [3] very similar to the lazy data-structures presented in this thesis. However, he seems to imply a stronger result. In our system, data structure slots are evaluated when a value is requested, but not necessarily required. In a lazy system, which Amamiya is claiming his to be (it appears, although his terminology is confusing), the contents of a data structure slot can be read and then thrown away without causing the evaluation of the delayed expression. We stress the importance of the distinction.

Amamiya also indicates that cells used to provide demand synchronization can be allocated statically and "... be free of the runtime memory allocation ...", implying static deallocation. But, this is not statically determinable in general. If we can tell when a delayed expression is evaluated, we can deduce that it is evaluated, and need not be delayed.

1.6 Overview of Thesis

In the remaining chapters of the thesis, we develop lazy data-structures, a limited form of lazy evaluation that extends the programming language Id, along with an implementation that naturally and efficiently employs architectural support. The language will not be as expressive as a lazy functional language, and we will consider the limitations.

In Chapter 2, we present Id#, Id plus lazy data-structures. We discuss the implementation in Chapter 3. Chapter 4 presents the programming methodology for using lazy data-structures, and discusses the expressive power and shortfalls. Chapter 5 concludes with discussions of the presented system and future work.

Chapter 2

Id#: A Language with Lazy Data-Structures

Id# extends Id [37] by introducing lazy data-structures[1]. Delayed expressions are always associated with data structure slots.

Most non-strict functional languages use a lazy evaluation rule as the default, and some allow the user to specify eager evaluation for certain function arguments. We take a converse approach, assuming eager evaluation, and allowing the producer of a data structure to indicate by explicit annotation that certain slots are to be assigned lazily. No annotation is required when a data structure is consumed; it is transparent to the consumer of a data structure whether the slot was assigned eagerly or lazily.

In this Chapter we present our approach to lazy evaluation, the syntax and semantics of our language, and the language-related (abstract) costs and benefits.

2.1 Approach

In Id#, the producer of a data structure can delay the evaluation of the expression that defines the contents of particular fields of records and the contents of individual slots of arrays. The consumer of a lazy data-structure does not know if a slot has been delayed, and has no way of telling if a slot has been delayed. The contents are automatically forced *if necessary*, and the computed value is stored for later use.

It is worth discussing what is meant by *if necessary.* A delayed expression resident in a data structure slot is evaluated whenever the contents are requested through a fetch, regardless of

[1]Lazy data-structures have already been incorporated into "current Id". We use "Id" to refer to the language with no lazy data-structures, and "Id#" to refer to the language *cum* lazy data-structures.

whether or not the value is actually thrown away or used in the computation. This point will be expanded presently by example.

Assignments to individual slots are annotated by the programmer for lazy evaluation, and all assignments to unmarked slots are evaluated eagerly.

As we have noted, Id is not a functional language. A large subset of Id is functional, though. The only non-functional construct in Id is the I-structure[2]. Note, however, that I-structures preserve the determinacy of Id. Although we are mostly interested in the functional subset of Id, we also include I-structures in Id# for completeness. The consequences the non-functional aspect will be discussed.

2.2 Syntax

There are four types of data structures in Id#: tuples, algebraic types[3] (including lists), arrays (these first three types are functional), and I-structures (non-functional). We consider syntax for creating each of these with lazy components. After discussing algebraic types, but before discussing arrays, we present two syntactic sugars for producing lists: *arithmetic sequences* and *list comprehension.*

2.2.1 Tuples

An expression in a tupling construct can be preceded by a "#" to indicate that it is to be delayed. In the following binding, the second and third tuple slots are delayed:

```
a_tuple = exp1, # exp2, # exp3, exp4;
```

Tuples are accessed by pattern matching, and lazy slots are evaluated when tuples are destructured. Consider the following definition that contains a let block that returns a three-tuple. Underscore ("_") is used in a destructuring pattern as a place holder when the value associated with a position is to be ignored.

```
consume_tuple =
  { e1,e2,_,e4 = a_tuple
   in
     e1,e2,e4};
```

[2]The term "I-structure" refers both to a language construct and to an implementation level construct. In this chapter, "I-structure" refers to the language construct, unless otherwise noted.

[3]Technically, tuples may be considered algebraic types. We separate them for expository purposes.

e1, e2, and e4 all need values, and, as a result, exp2 is evaluated, while exp3 remains delayed. Even if e2 did not appear in the return expression or if e2 were not used by the caller of consume_tuple, exp2 would be evaluated.

2.2.2 Algebraic Types

The other heterogeneous, unsubscripted types are collectively called algebraic types. Cons (the list constructor) is an example of a built-in algebraic type. Conses are constructed with an infix ":". By using the "#" annotation, either the head or tail (or both) can be delayed. Consider the following code:

```
some_conses =
  { c1 =  exp1 :  exp2;
    c2 =  exp3 : #exp4;
    c3 = #exp5 :  exp6;
    c4 = #exp7 : #exp8
   in
     c1,c2,c3,c4};
```

c1 is a normal cons: both the head and tail are evaluated eagerly. c2 has an eager head and a lazy tail, and c3 has a lazy head and an eager tail. c4 is a lazy cons: both the head and tail are evaluated lazily.

Eager and lazy slots of cons cells are accessed uniformly by pattern matching, oblivious to the method of assignment. When a lazily assigned slot is accessed, the computation is performed and the value is returned. Consider the following binding:

```
consume_conses =
  { c1,c2,c3,c4 = some_conses;
    e3:e4 = c2;
    _ :e6 = c3;
    e7:_  = c4
   in
     e3,e4,e6};
```

e3, e4, e6, and e7 all need values, and, as a result, exp4 and exp7 are evaluated (even though e7 is not returned), but exp5 and exp8 remain delayed.

Similarly, user-defined algebraic types can have lazy slots. Expressions in algebraic type constructors can be preceded by a "#". In the following code, the foo type is introduced with two constructors, Bar and Baz. The call to make_complex_foo, which is destined for the first slot of a Baz type foo, is delayed. "#" binds less tightly than procedure application.

```
type foo = Bar | Baz foo N;

Baz (# make_a_complex_foo a b c) 17
```

The first component of the above expression is delayed. This technique is especially useful for defining recursive data structures. And, once again, algebraic types are accessed using pattern matching, with delayed slots being forced implicitly on selection.

Since cons is such an important constructor, we introduce special syntax for delaying its arguments. The "#" that would normally precede the first argument to cons may succeed it as follows:

```
same_conses =
  { c1 = exp1  :  exp2;
    c2 = exp3  :# exp4;
    c3 = exp5 #:  exp6;
    c4 = exp7 #:# exp8
   in
     c1,c2,c3,c4};
```

This variation allows us to think of four infix cons operators: normal cons (":"), stream cons (":#", *i.e.*, tail-lazy cons), head-lazy cons ("#:"), and head-lazy stream cons ("#:#", *i.e.*, fully-lazy cons).

2.2.3 Arithmetic Sequences and List Comprehension

Id# has special syntax for generating lists called *arithmetic sequences* and *list comprehension.* Arithmetic sequences (like Miranda's dotdot notation [44]) are convenient ways of expressing a range of integers. List comprehension (based on Miranda's list comprehension [44]) provides a compact way of specifying more complex lists. Arithmetic sequences and list comprehension are based on Zermelo-Frankel set notation.

Arithmetic Sequences

Arithmetic sequences are language idioms which denote ascending or descending lists of numbers with a constant first-difference. Consider the arithmetic sequence for generating the list of integers from one to ten, and the arithmetic sequence for generating the list of integers from ten to one. Each binding is followed by a pseudo-Id expression that gives its meaning (after the "%="). "%" is Id's comment character. ":" associates to the right.

```
typeof one2ten = list N;
one2ten = 1 to 10;
%= 1 : 2 : 3 : ... : 10 : nil

typeof ten2one = list N;
ten2one = 10 downto 1;
%= 10 : 9 : 8 : ... : 1 : nil
```

The "`<exp> to <exp> by <exp>`" idiom allows arithmetic sequences with a computed first difference to be specified conveniently.

```
typeof odds_between_1_and_20 = list N;
odds_between_1_and_20 = 1 to 20 by 2;
%= 1 : 3 : 5 : ... : 19 : nil
```

These behaviors can be captured in Id# "library routines" as follows. An "_" is prefixed to all functions to preclude confusion with any keywords.

```
typeof _to = N -> N -> (list N);
def _to lo hi =
  if lo > hi then
    nil
  else
    lo : _to (lo+1) hi;


typeof _to_by = N -> N -> N -> (list N);
def _to_by lo hi step =
  if lo > hi then
    nil
  else
    lo : _to_by (lo+step) hi step;
```

These facilities existed in Id. Now we extend them to allow us to conveniently express infinite arithmetic sequences. **upfrom** `<exp>` and **downfrom** `<exp>` are new idioms that denote infinite arithmetic sequences of integers, the former ascending, and the latter descending. Consider the following binding for the integers. ":#" associates to the right.

```
typeof ints = list N;
ints = upfrom 1;
%= 1 :# 2 :# 3 :# ...
```

Both upfrom and downfrom can be used in conjunction with the by keyword to vary the step size of the sequence. Consider the stream of odd integers:

```
typeof odds = list N;
odds = upfrom 1 by 2;
%= 1 :# 3 :# 5 :# ...
```

These behaviors can be expressed in Id#. Each succeeding recursion is buried in the delayed tail of a stream cell. Consider the following definitions for _upfrom and for _upfrom_by. Definitions for _downfrom and for _downfrom_by are similar.

```
typeof _upfrom = N -> (list N);
def _upfrom n = n :# _upfrom (n+1);

typeof _upfrom_by = N -> N -> (list N);
def _upfrom_by n step = n :# _upfrom_by (n+step) step;
```

Even though the "by expression" can be variable, it is constant with respect to the sequence; it is evaluated once, and the value is used repeatedly.

Infinite arithmetic sequences point to a new opportunity, the possibility of performing speculative computation. Each time we reach "the current end" of an infinite arithmetic sequence, we can extend it by more than one element. The unwind keyword gives the programmer control over the amount that the stream expands each time its tail is forced. Consider the following binding for the integers that expands three slots at a time.

```
typeof ints_unwind3 = list N;
ints_unwind3 = upfrom 1 unwind 3;
%= 1 : 2 : 3 :# 4 : 5 : 6 :# ...
```

Abstractly, if we are unwinding an infinite arithmetic sequence by three, each time the delayed tail of the sequence is reached, three elements are produced. By default, arithmetic sequences unwind by one.

```
typeof _upfrom_unwind = N -> N -> (list N);
def _upfrom_unwind lo unwind =
  {def _upfrom_unwind_ lo count =
        if count == 1 then
          lo :# _upfrom_unwind_ (lo+1) unwind
        else
          lo :  _upfrom_unwind_ (lo+1) (count-1);
      in
        _upfrom_unwind_ lo unwind};
```

It is possible to apply unwinding to finite arithmetic sequences as well, but there is a small complication, as the unwinding may not be finished when the list ends. Suppose we are unwinding a finite arithmetic sequence by three elements at a time. Each time the delayed tail of the stream is reached, up to three elements are produced. Fewer than three elements are produced if the list ends, as we see in the following example:

```
typeof one2four_unwind3 = list N;
one2four_unwind3 = 1 to 4 unwind 3;
%= 1 : 2 : 3 :# 4 : nil
```

Unwinding can be used in combination with the by keyword. Id definitions for `_to_unwind`, `_downto_unwind`, `_to_by_unwind`, *etc.*, are omitted.

All of the previous "library routines" were defined in terms of recursive procedures. Whenever unwinding is present, a routine can be defined more efficiently using Id#'s looping construct in conjunction with a data structure called an "open list". Open lists (similar to Prolog's "difference lists" [17]) use I-structures to define a list by successively appending elements to the "open slot" at the end[4]. Arithmetic sequences that unwind one element at a time such as `upfrom 1` do not benefit from this opportunity.

List Comprehension

List comprehension is based on Zermelo-Frankel set notation and allows the programmer to conveniently express lists that are generated by mapping and filtering over other lists.

In a list comprehension, an expression is evaluated in a sequence of binding environments, and the results are collected in order. A list comprehension begins with "{:". The following list comprehension denotes the list of integers from one to ten.

```
typeof one2ten = list N;
one2ten = {: i || i <- 1 to 10};
%= 1 : 2 : 3 : ... : 10 : nil
```

[4]Examples of the use of open lists as well as additional discussion can be found in [6].

i <- 1 to 10 generates a sequence of binding environments in which i is bound to 1, 2, ..., 10 and is called a generator.

More generally, *any* expression can be evaluated in the sequence of generated binding environments. ":" binds less tightly than procedure application.

```
{: f i || i <- is}
%= (f i1) : (f i2) : (f i3) : ... : (f in) : nil
%=  f i1  :  f i2  :  f i3  : ... :  f in  : nil
```

where ij is the jth element of the list is.

Filtering can be accomplished by associating the when or unless keywords with a generator. The following list comprehension, which assumes the existence of a predicate to test integers for primality, denotes the list of primes below twenty:

```
typeof prime? = N -> B;
typeof primes_below_20 = list N;
primes_below_20 = {: i || i <- 1 to 20 when prime? i};
%= 2 : 3 : 5 : ... : 19 : nil
```

Several generators may be present, in which case the sequence of binding environments is given by a row major order traversal of the cross product of the generated environments. Each inner environment can use names defined in an outer scope. The following list comprehension denotes the lower right triangle of a three-by-three grid:

```
typeof grid = list (N,N);
grid = {: (i,j) || i <- 1 to 3 & j <- 1 to i};
%= (1,1):(2,1):(2,2):(3,1):(3,2):(3,3):nil
```

Just as the generation of lists is supported with list comprehension syntax, the generation of streams is supported with stream comprehension syntax. A stream comprehension begins with "{:#", as a stream is a tail-lazy list, and ":#" is tail-lazy cons (stream cons). Consider the following stream comprehension for the stream of integers from one to ten:

```
typeof one2ten = list N;
one2ten = {:# i || i <- 1 to 10};
%= 1 :# 2 :# 3 :# ... :# 10 :# nil
```

As in list comprehension, *any* expression can be evaluated in the sequence of generated binding environments. ":#" binds less tightly than procedure application.

```
{:# f i || i <- is}
%= f i1 :# f i2 :# f i3 :# ... :# f in :# nil
```

where ij is the jth element of the list is.

As in list comprehension, filtering can be accomplished by associating the when or unless keywords with a generator. The following stream comprehension denotes the stream of primes below twenty:

```
typeof prime? = N -> B;
typeof primes_below_20 = list N;
primes_below_20 = {:# i || i <- 1 to 20 when prime? i};
%= 2 :# 3 :# 5 :# ... :# 19 :# nil
```

Stream comprehension offers several opportunities that are not available to list comprehension. The first new opportunity is the application of program controlled unwinding. Consider the following definition which maps a function over a stream, and unwinds two elements at a time:

```
typeof smap1_unwind2 = (N->N) -> (list N) -> (list N);
def smap1_unwind2 f is = {:# f i unwind 2 || i <- is};
%= f i1 : f i2 :# f i3 : f i4 :# ... :# f in : nil
```

By default, a stream comprehension unwinds by one. Suppose we are unwinding a stream comprehension by three elements at a time. Each time the delayed tail of the stream is reached, up to three elements are produced. Fewer than three elements are produced if the sequence of binding environments ends, as we see in the following example:

```
typeof one2four_unwind3 = list N;
one2four_unwind3 = {:# i unwind 3 || i <- 1 to 4};
%= 1 : 2 : 3 :# 4 : nil
```

Variable unwinding is also possible:

```
typeof ints_with_varied_unwinding = list N;
ints_with_varied_unwinding = {:# i unwind i || i <- 1 to 10};
%= 1 :# 2 : 3 :# 4 : 5 : 6 : 7 :# : 8 : 9 : 10 : nil
```

Exactly in which binding environment the unwinding expression is evaluated is very important, as the value of the "unwind expression" can change. The "unwind expression" is evaluated in *the first* binding environment, and reevaluated each time the stream suspends and resumes. In other words, the "unwind expression" is evaluated once for each "forced tail".

A new type of unwinding is also possible. The `while` and `until` keywords allow the stream to unwind while or until a predicate is satisfied. The following stream denotes the integers from one to ten, and unwinds until it finds a prime element:

```
typeof prime_unwinder = list N;
prime_unwinder = {:# i until prime? i || i <- 1 to 10};
%= 1 : 2 :# 3 :# 4 : 5 :# 6 : 7 :# : 8 : 9 : 10 : nil
```

The expression associated with `while` or `until` is evaluated in *every* binding environment. The predicate acts as a post-test, *i.e.*, it specifies whether the next element should be produced eagerly. At least one stream element is produced, regardless of the unwinding controls, unless the generators run out of binding environments.

Numerical unwinding (using the `unwind` keyword) and boolean unwinding (using the `while` and `until` keywords) can be combined, in which case the stream suspends if either the unwind count dips below one, or the boolean test indicates that a suspension is in order.

A list comprehension can be viewed as a stream comprehension with infinite unwinding. Conversely, a stream comprehension can end up producing a data structure of finite length, as does a list comprehension.

The most interesting new opportunity still remains: a stream comprehension can have infinite generators. Consider the following stream comprehension for the squares of the integers, which has an infinite arithmetic sequence as a generator:

```
typeof squares = list N;
squares = {:# i^2 || i <- upfrom 1};
%= 1 :# 4 :# 9 :# ...
```

We could enumerate the first octant (the grid points with integer coordinates in the first quadrant, above and including the x-axis and below the 45° line) as follows:

```
typeof octant1 = list (N,N);
octant1 = {:# (x,y) || x <- upfrom 1 & y <- 0 to x-1};
%= (1,0) :# (2,0) :# (2,1) :# ...
```

Me might try naively to enumerate the first quadrant similarly:

```
typeof quadrant1 = list (N,N);
quadrant1 = {:# (x,y) || x <- upfrom 1 & y <- upfrom 0};
%= ???
```

Diagonalization (*i.e.*, fairly producing the cross-product) is not automatic. The stream quadrant1 will climb the grid along the vertical line $x = 1$, and never enumerate any other points.

```
quadrant1 = (1,0) :# (1,1) :# (1,2) :# (1,3) :# ...
```

Conversely, Miranda provides an idiom for automatic diagonalization. In Miranda, brackets ("[" and "]") set off lists, and two dots ("..") indicate an integer range.

```
quadrant1 = [(x,y) // x<-[1..]; y<-[0..]]
%= [(1,0),(1,1),(2,0),(1,2),(2,1),(3,0), ... ]
                                                    Miranda
```

In the following example, the unwind expression is variable:

```
typeof octant1_a_column_at_a_time = list (N,N);
octant1_a_column_at_a_time =
  {:# (x,y) unwind x || x <- upfrom 1 & y <- 0 to x};
```

Each time we consume a vertical column of the triangle, the next column is computed eagerly. The unwinding facility is thus a tool for speculative computation.

Two more variants of list comprehension are of interest: head-lazy list comprehension, and head-lazy stream comprehension, corresponding to the the following analogy: ":":"#:" :: ":#":"#:#". Put another way, head-lazy list comprehension is the head-lazy version of list comprehension, and head-lazy stream comprehension is the head-lazy version of stream comprehension. Head-lazy lists are built with cons cells with the head being evaluated lazily and tail being evaluated eagerly. Head-lazy streams are built with cons cells with both the head and tail being evaluated lazily.

A head-lazy list is a list where the spine (skeleton) of the list is expanded eagerly, but the elements are only computed on demand. The head-lazy list comprehension, like the list comprehension with the initial ":" being replaced by "#:", is a convenient way to express head-lazy lists. In the following code, the spine is expanded immediately, but the function f is not applied to any elements until they are requested by a consumer:

```
{#: f x || x <- 1 to 10}
%= f 1 #: f 2 #: ... #: f 10 #: nil
```

Suppose the function f is very expensive to compute; the list produced is short; we only need some list elements; but we cannot tell in advance which elements are needed. The head-lazy list comprehension is designed for this scenario, the "expensive slots" programming paradigm.

A head-lazy stream is one in which both the elements and the spine are expanded on demand. The head-lazy stream comprehension, like the stream comprehension with the initial ":#" being replaced by "#:#", is a convenient way to express head-lazy streams. In the following code, the head-lazy stream is lazy in the head and tail. The function f is not applied to any elements that are not needed, and the tail expands on demand.

```
{#:# f x || x <- upfrom 1}
%= f 1 #:# f 2 #:# f 3 #:# ...
```

Suppose the function f is very expensive to compute; we need only a fraction of some finite prefix of the elements; but we cannot tell in advance which elements are needed. The head-lazy stream comprehension is designed for this scenario, which combines the "expensive slots" and "infinite structures" programming paradigms.

The properties of lists (streams) produced by any of these varieties of list comprehension are determined by the type of the constructor, not by the type of an embedded generator. A list (as opposed to a stream) with an infinite generator will diverge, as we see in the following example:

```
typeof diverge = list N;
diverge = {: x || x <- upfrom 1};
%= 1 : 2 : 3 : ...
```

All the facilities described in this section can be mixed and matched. As comprehension is syntactic sugar, all combinations can be expressed directly in Id. This is most easily done in terms of recursive procedures, but, as we indicated at the end of the previous section, more efficient techniques are available. Streams with unwinding as well as streams with filters (**when** and **unless**) are prime candidates for loop-style implementations. Unwinding can proceed eagerly until it is time to suspend. If an environment is discarded by a filter, the next one can be generated eagerly. An array of optimizations are possible, and a few are described in the remainder of this section.

If a generator is an arithmetic sequence, no intermediate stream need be generated. The state of the environment can be passed directly from one recursion (iteration) to the next. This is especially important when generating cross products, and many streams can be avoided.

Lists that are passed in (a <- as) and finite generators require a test for the end. Some generators, however, are known to be infinite, and an end test can be avoided.

Arithmetic sequences and list comprehension provide support mainly for the "progıamming with infinite data-structures structures" paradigm. The next two section, on arrays and I-structures address the "programming with expensive slots" paradigm.

2.2.4 Arrays

In Id#, an array is functional indexed data-structure. Each slot of an array may be assigned at most once, and arrays can be constructed with lazy slots.

Arrays are produced using *array comprehension.* An array comprehension declares the bounds of the array and has clauses that specify the elements for regions of the array. Both the index expression and the actual expression are evaluated in the specified sequence of binding environments. In the following array comprehension, the array is filled with integers in descending order:

```
typeof a1 = array N;
a = {array (1,n) | [i] = n-i+1 || i <- 1 to n}
```

The index expression can also be non-trivial. The following array comprehension produces an array identical to the preceding one:

```
typeof a2 = array N;
a2 = {array (1,n) | [n-i+1] = i || i <- 1 to n}
```

Several clauses can be given to fill several regions. The procedure **identity_matrix** uses array comprehension to produce the identity matrix of size n by n:

```
typeof identity_matrix = N -> (I_matrix N);
def identity_matrix n =
  {matrix ((1,n),(1,n))
   | [i,j] = 0 || i <- 1 to n-1 & j <- i+1 to n    % above the diagonal
   | [i,i] = 1 || i <- 1 to n                      % the diagonal
   | [i,j] = 0 || i <- 2 to n   & j <- 1 to i-1};  % below the diagonal
```

To have a lazy slot, "#" replaces "=" as follows:

```
typeof edge = N -> N;
typeof middle = N -> N;
typeof an_array = N -> (array N);
def an_array n =
  { array (1,n)
   | [1] = edge 1
   | [i] # middle i || i <- 2 to n-1
   | [n] # edge n };
```

Slot 1 is assigned eagerly, and the rest of the slots are computed only after being selected by an array selection operation.

```
typeof consume_array = N -> N -> N;
def consume_array n i =
  { a = an_array n
   in
     a[i]};
```

2.2.5 I-structures

I-structures, as we have noted, are non-functional data structures. They are indexed and may have distributed definitions. Each slot of an I-structure may be assigned at most once. I-structure slots behave something like Prolog's "logic variables"[4]. I-structures, like all data structures in Id, can be constructed with lazy slots.

I-structures can be created in one place and filled in any number of places, very much like Fortran arrays. I-structures slots are assigned lazily by, once again, replacing the "=" by "#". `an_I_structure` has a *for loop* that is run purely for side effect[5]. Note how the loop index `i` is generated from an arithmetic sequence in the "comprehension style".

```
typeof an_I_structure = N -> (I_array N);
def an_I_structure n =
  { a = I_array (1,n);
    a[1] = edge 1;
    { for i <- 2 to n-3 do a[i] # middle i};
    a[n] # edge n
   in
     a};
```

[5]Id# loops can also return a result.

Slot 1 is assigned eagerly, slots 2 through n-3 and the last slot are assigned lazily, and slots n-2 and n-1 are left unassigned. The contents of the delayed slots are computed only after being selected by an I-structure selection operation. In the following code, slots n-2 and n-1 are filled in, the former, eagerly, and the latter, lazily. Then a slot is selected, which may or may not cause a delayed expression to be evaluated, depending on whether or not the selected slot was assigned lazily or eagerly. The let expression in **consume_I_structure** contains one binding, and two I-structure assignments.

```
typeof consume_I_structure = N -> N -> N;
def consume_I_structure n i =
  { a = an_I_structure n;
    a[n-2] = middle (n-2);
    a[n-1] # middle (n-1)
   in
     a[i]};
```

2.3 Operational Semantics

A small extension to the rewrite rules found in [9] allows us to capture the operational semantics of our new constructs. Rewrite rules are described in a two column format. The left column corresponds to an expressions and its binding environment (listed below the expression), and the right side is the rewritten expression along with its bindings. The expression may change as may the bindings, and new bindings may be introduced.

```
expression                        ⇒   expression'
binding1 ; ... ; bindingn             binding1 ; ; ... ; bindingm
```

The rewrite rules given in [9] are context sensitive. For example, a sub-expression may not be evaluated "*within the* **Then** *or* **Else** *arms of a conditional expression*". The rewrite rules may not look inside certain expressions, and we make this behavior explicit by enclosing any *opaque expressions* in double quotes (""). When a conditional is first written down, the consequent and alternate clauses are surrounded by quotes and are opaque. After the boolean value is determined, the conditional is contracted, the quotes are removed, and the rules are allowed to perform reductions within the previously protected expression. A conditional is rewritten as follows:

```
(If true Then "E1" Else "E2")       ⇒   (E1)
B1 ; ... ; Bn                           B1 ; ... ; Bn
```

We only need one rewrite rule to support all of our constructs. First, we note that all expressions preceded by a "#" start off as opaque (enclosed in double quotes). Whenever an identifier is selected as a redex, if it is bound to an opaque expression the quotes are removed. An identifier may not be taken as a redex if it is within an opaque expression, or in a data structure.

```
X                                      X
B1 ; ... ; Bn ; X = "E1"        ⇒      B1 ; ... ; Bn ; X = E1
```

This rule applies to conses, tuples, algebraic types, arrays, and I-structures.

Consider the following definition:

```
typeof ints_from = N -> (list N);
def ints_from n = n : #ints_from (n+1);
```

As an example rewrite sequence, consider the evaluation of an expression. We begin with a query for the second element of a stream and an empty environment. Expressions that are underlined are about to be rewritten.

```
hd (tl (ints_from 1))
<no bindings>
```

The application of ints_from is expanded, introducing formal parameter n1. A binding for n1 appears in the binding list. Note the opaque expression: we substitute for the exposed n1, but the n1 buried in the opaque expression may not be rewritten.

```
⇒
hd (tl (n1 :  "ints_from (n1+1)"))
n1 = 1
⇒
hd (tl (1 :  "ints_from (n1+1)"))
n1 = 1;
```

Apply the cons, introducing new names for the parts.

```
⇒
hd (tl <cons x001 x002>)
n1 = 1; x001 = 1; x002 = "ints_from (n1+1)"
```

Apply the tail function.

```
⇒
hd x002
n1 = 1; x001 = 1; x002 = "ints_from (n1+1)"
```

Use the new rule to expose the opaque expression.

```
⇒
hd x002
n1 = 1; x001 = 1; x002 = ints_from (n1+1)
```

Apply ints_from again, introducing formal parameter n2 and another opaque expression.

```
⇒
hd x002
n1 = 1; x001 = 1; x002 = n2 :  "ints_from (n2+1)"; n2 = n1+1
⇒
hd x002
n1 = 1; x001 = 1; x002 = n2 :  "ints_from (n2+1)"; n2 = 1+1
⇒
hd x002
n1 = 1; x001 = 1; x002 = n2 :  "ints_from (n2+1)"; n2 = 2
```

Substitute for the exposed n2. As before, the n2 buried in the opaque expression may not be rewritten.

```
⇒
hd x002
n1 = 1; x001 = 1; x002 = 2 :  "ints_from (n2+1)"; n2 = 2
```

Applying another cons operator, we introduce two more new names.

```
⇒
hd x002
n1 = 1; x001 = 1; x002 = <cons x003 x004>; n2 = 2;
x003 = 2; x004 = "ints_from (n2+1)"
```

Lookup x002 in the environment.

```
⇒
hd <cons x003 x004>
n1 = 1; x001 = 1; x002 = <cons x003 x004>; n2 = 2;
x003 = 2; x004 = "ints_from (n2+1)"
```

Apply the **head** function.

```
⇒
x003
n1 = 1; x001 = 1; x002 = <cons x003 x004>; n2 = 2;
x003 = 2; x004 = "ints_from (n2+1)"
```

Finally, lookup x003 in the environment.

```
⇒
2
n1 = 1; x001 = 1; x002 = <cons x003 x004>; n2 = 2;
x003 = 2; x004 = "ints_from (n2+1)"
```

Although there are slight variations on the reduction order for the above example, there are no essential differences from the preceding reduction order.

Removing the quotes and exposing an opaque expression is analogous to enabling the evaluation of a delayed expression. According to our new rewrite rule, a delayed expression is evaluated at most once, as soon as its value is needed.

Arithmetic sequences and stream comprehension are syntactic sugars and are thus defined within the language. They need no special treatment here.

2.4 Abstract Costs and Benefits

There are several abstract costs to our approach. The first is expressive clarity. What are the repercussions of sprinkling our programs with hash marks? Can annotated programs be understood clearly and intuitively? Next is expressive power. Can we code all the programs we would like in a straightforward manner? This is an obvious question, since we cannot express all the programs that can be expressed in a lazy functional language. We consider these potential problems in detail in Chapter 4.

Are there advantages to explicit annotations? Laziness is expensive, and annotations prevent us from ignoring the expense by sweeping it under the rug.

The first and foremost advantage of our approach is the fact that it adds a power to the language that it did not formerly possess. We can now express infinite data-structures and data structures with expensive slots directly.

2.5 Summary

This chapter extends Id to incorporate lazy data-structures. When constructing any data structure in Id#, the expression destined for any slot may be annotated with a "#" to indicate that the evaluation of an expression should be delayed until a consumer requests the value of the slot. Using lazy data-structures, we can write programs using both the infinite data-structures programming paradigm and the data structures with expensive slots programming paradigm.

Arithmetic sequences and stream comprehension provide special syntax for stream programming. Explicit control over unwinding allows the programmer to express speculative computation.

Chapter 3

Implementation of Id#

In order to implement the language extensions described in the previous chapter, support is required both in compiler and architecture. In this chapter we describe our approach to implementation, support from the architecture, compilation techniques, and the concrete costs and benefits of our approach.

3.1 Approach

The compiler generates dataflow graphs to build thunks which embed the delayed expressions, and the architecture provides synchronization for triggering the delayed expressions and reading the results. We have already presented the language extensions, *i.e.*, the system at the highest level. Now we present the support structure from the bottom up. First, we present the architectural extensions that provide the necessary hardware support. Then, we present the compiler extensions needed to reduce our language to architectural primitives.

3.2 Architectural Extensions

Several architectural extensions are needed to support lazy data-structures. First, we extend I-structures[1] [10, 23] to *L-structures* (lazy structures), which provide the synchronization required to support lazy data-structures. Next, we provide support for *suicide procs*, procedures embedding delayed expressions that are invoked by the memory system rather than by an explicit procedure call. And finally, we introduce a new *manager* (dataflow run-time system call) to invoke suicide procs.

[1]Unless otherwise noted, in this chapter the term "I-structure" refers to the an implementation mechanism, not a language construct.

3.2.1 L-structures

Tuples, algebraic types, I-structures (the language "I-structure"), and arrays will all be implemented using L-structures. An L-structure is like an array in any programming language, but each slot has synchronization built into it. L-structures are a variation on I-structures [9]. We begin our development of L-structures by first presenting I-structures.

Each slot of an I-structure has status bits associated with it. If a consumer arrives before the producer, the `fetch` is remembered locally by the I-structure slot in a list until the value is `stored`. The state of the slot is recorded by the status bits. Figure 3.1 is the state transition diagram for an *individual I-structure slot* (as opposed to the entire I-structure). Activity caused by a transition is indicated after the slash.

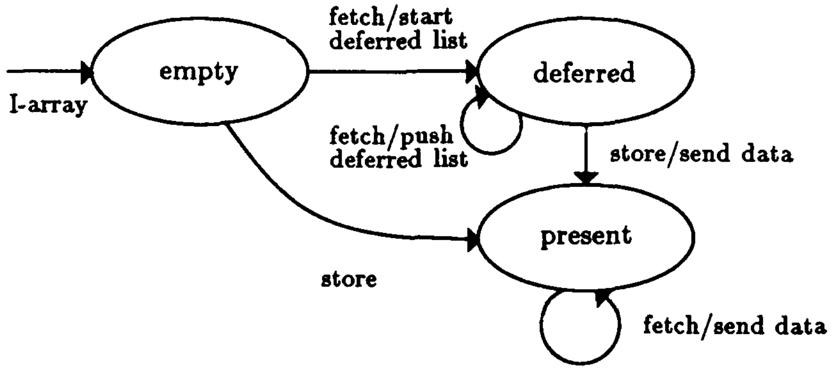


Figure 3.1: State Transition Diagram for an I-structure Slot

There are three operations on I-structures, as we have seen in Figure 3.1. `I-array` takes an integer argument and returns an empty I-structure of corresponding size. `store` places data in an empty slot, satisfying any deferred fetches. `fetch` returns the data if it is *present*; otherwise, it registers a deferred fetch. These operations are summarized in Table 3.1.

`I-array size`	⇒	`<I-structure-address>`
`store <I-structure-address> value`	⇒	`<acknowledgment>`
`fetch <I-structure-address> destination`	⇒	`value`

Table 3.1: I-structure Operations

Two paths through the state transition diagram are possible. Both paths are demonstrated in parallel in the following simulation:

Simulation Step 1: Allocate **x**, an I-structure of size two. Both slots start in the *empty* state. The state of each slot appears below the data.

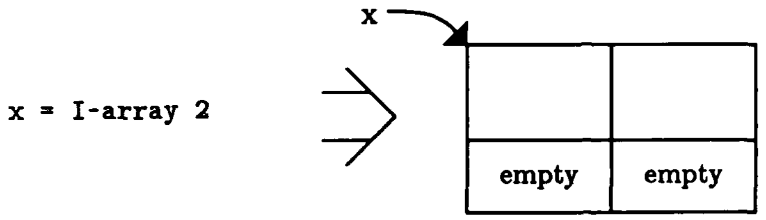


Figure 3.2: Simulation Step 1: Allocate an I-structure of Size Two

Simulation Step 2: Simultaneously `store` the number 5 in the first slot `x[0]` (I-structures are zero-indexed) and `fetch` the contents of `x[1]` for reader `r1`. `x[0]` enters the *present* state as it now holds valid data, and `x[1]` enters the *deferred* state and contains a pointer to the deferred fetch list containing `r1` (the slash in the right half of the deferred fetch list indicates the end of the list).

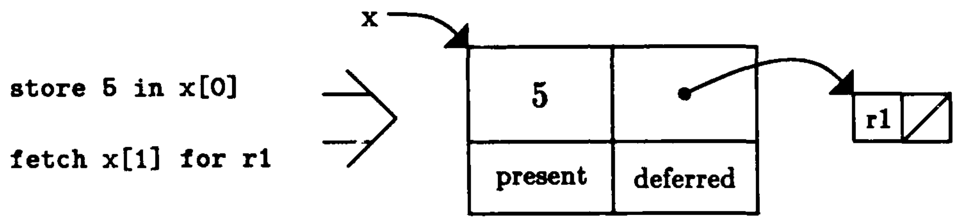


Figure 3.3: Simulation Step 2: `store` in x[0], `fetch` from x[1]

Simulation Step 3: `fetch` the contents of `x[0]` for reader `r2` and `fetch` the contents of `x[1]` for `r3`. Since `x[0]` has valid data, `r2` is sent the stored value 5. Since `x[1]` has already been deferred (no data), `r3` is pushed onto the deferred fetch list for `x[1]`.

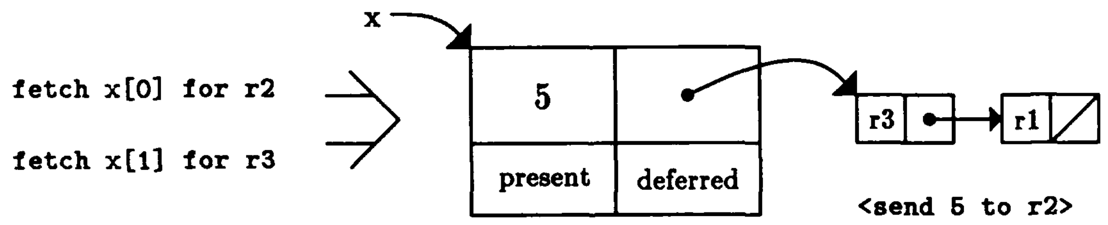


Figure 3.4: Simulation Step 3: `fetch` from x[0], `fetch` from x[1]

Simulation Step 4: `store` the value 10 in `x[1]`. `x[1]` makes the transition to the *present* state as the deferred fetches are satisfied by sending the value to `r3` and `r1`.

The two slot I-structure we have been modeling could have been a 2-tuple, a cons cell, an I-structure of size two (the language "I-structure"), or a variety of other data structures. All data structures look the same at this level: in Id, all data structures are implemented using

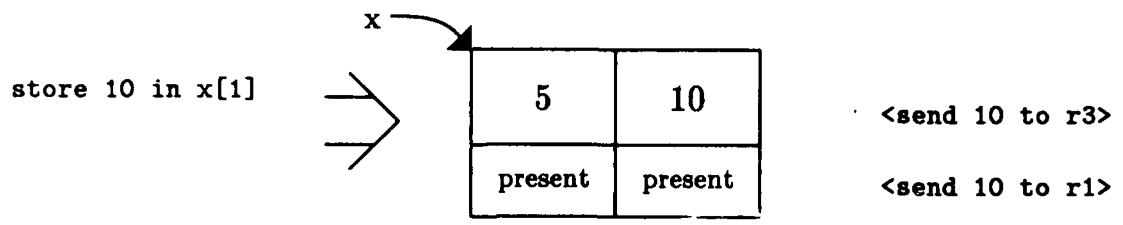


Figure 3.5: Simulation Step 4: **store** in x[1]

I-structures, which provide the low-level synchronization needed to support producer/consumer synchronization in a multiprocessor.

Now we introduce L-structures to provide a substrate for implementing lazy data-structures. L-structures provide support for producer/consumer synchronization and, additionally, for explicitly delaying and implicitly evaluating the contents of individual slots.

I-structures synchronize readers and writers; readers are stalled until the value is written. L-structures also synchronize readers and writers, and, in addition, they synchronize the evaluation of delayed expressions. Evaluation is stalled until there is a reader (a request for the value), and readers are stalled until the value arrives.

L-structures have four memory operations: **L-array**, **store-thunk**, **store-data**, and **fetch**. **L-array** takes an integer argument and returns an empty L-structure of corresponding size; **store-thunk** places a thunk in an empty slot; **fetch** *ejects* any thunk that is present, causing the thunk to be evaluated, and registers a deferred fetch; and **store-data** places data in a slot, satisfying any deferred fetches. L-structure operations are summarized in Table 3.2.

L-array size	⇒	**<L-structure-address>**
store-thunk <L-structure-address> thunk	⇒	**<acknowledgment>**
store-data <L-structure-address> value	⇒	**<acknowledgment>**
fetch <L-structure-address> destination	⇒	**value**

Table 3.2: L-structure Operations

When used to delay computation, an L-structure slot is "written twice": first with the thunk, then with the value the thunk computes. We add the **delayed** and **evaluating** states to the I-structure state transition diagram to achieve lazy behavior, as shown in Figure 3.6. Figure 3.6 is the state transition diagram for *each slot* of an L-structure.

Five "memory snapshots" of a single slot of an L-structure corresponding to the five states of Figure 3.6 are arranged in Figure 3.7. The state of the slot appears in the lower box, and the

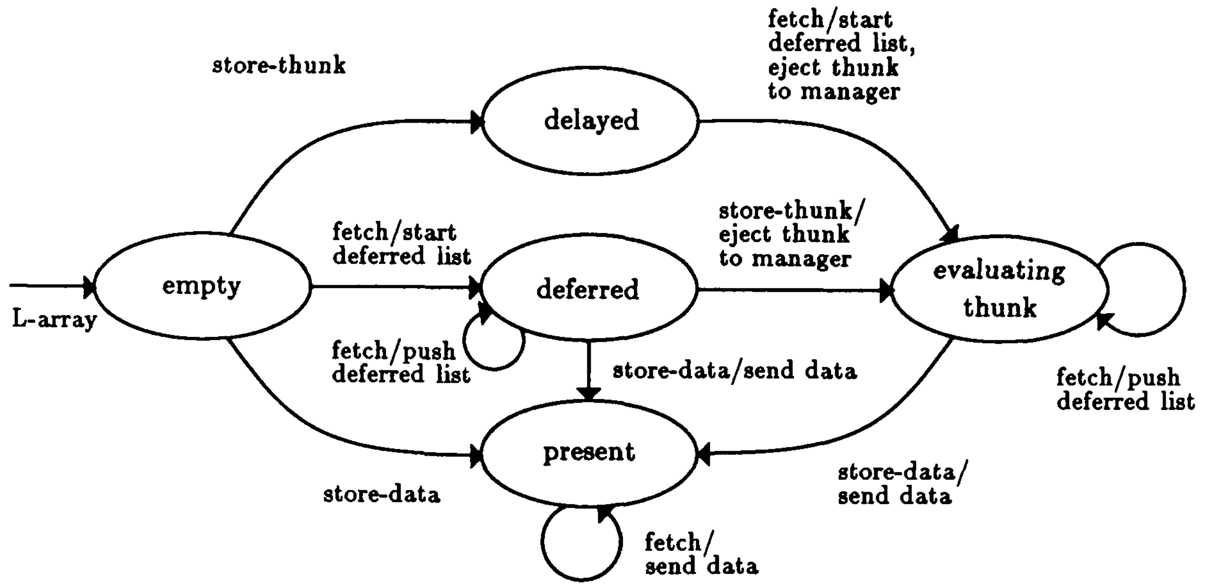


Figure 3.6: State Transition Diagram for an L-structure Slot

data can be found in the upper box. The slot starts in the *empty* state (on the left). Suppose a `store-thunk` operation is the first to arrive. The thunk (delayed expression and environment) is stored in the slot as shown at the top. Note the pointer from the thunk back to the delayed slot. When a `fetch` operation arrives, the delayed expression is spawned (by magic, for now). The star-burst at the lower right represents the spawned process. The `fetch` is put on a deferred fetch list, along with any other `fetches` that arrive. `r1` and `r2` represent readers of the slot that have been deferred. Eventually, the spawned computation will finish evaluating the delayed expression and issue a `store-data` for this slot. The value of the previously delayed expression is stored, and deferred fetches are satisfied.

Suppose a `fetch` reaches the slot first. The status of the delayed slot starts in the *empty* state, moves to the *deferred* state, where more `fetches` may be queued, to the *evaluating* state, where the delayed expression is spawned, and down to the *present* state, where the value of the previously delayed expression sits, awaiting other readers.

L-structure operations are a superset of (and consistent with) I-structure operations. Correspondingly, the L-structure transition diagram embeds the I-structure transition diagram, and as a result, I-structure behavior can also be achieved in an L-structure.

There are two remaining paths through the snapshots which correspond to I-structure behavior (no delayed computation, just producer/consumer synchronization). If a `fetch` arrives before the `store-data`, the status of the slot moves from the *empty* state to the *deferred* state

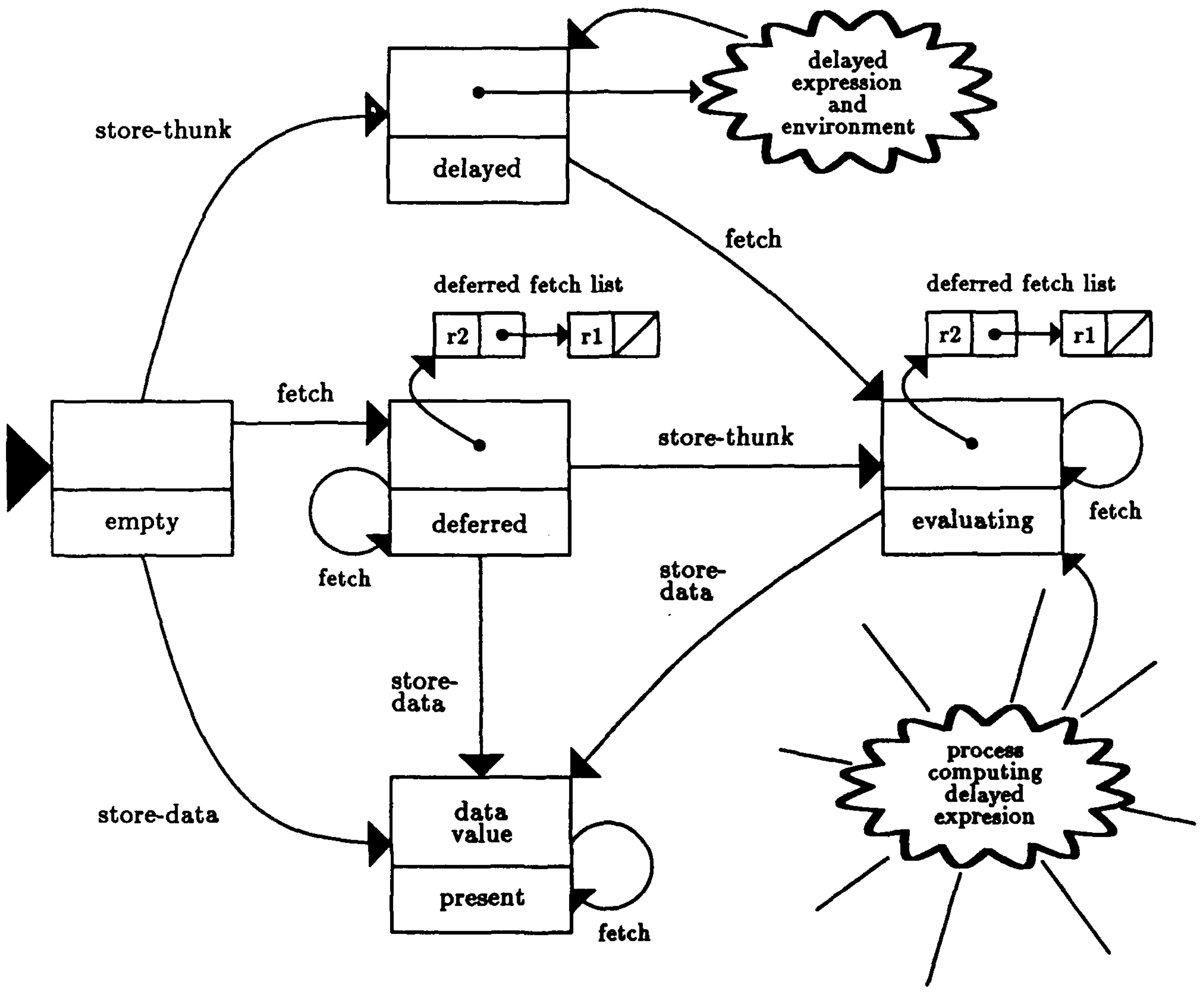


Figure 3.7: State Transition Memory Snapshots

where more readers may be deferred, and down to the *present* state when the data arrives on a `store-data` operation. If the `store-data` precedes any fetches, the status of the slot simply moves from the *empty* state to the *present* state, where the data waits for any readers.

In Id#, tuples, algebraic types, arrays, and I-structures (the language "I-structure") can all be implemented using L-structures.

Minimal changes are needed in the TTDA's I-Structure Memory Controller [23] to accomplish the desired new behavior on the TTDA. Also, this behavior is simple to achieve on Monsoon [38].

3.2.2 Thunks

A thunk embeds the delayed expression and its environment. The delayed expression is embedded (at compile time) in a dataflow graph that fetches its free variables from the environment and stores the result in the slot containing the thunk. This dataflow graph (along with the dataflow graph to compute the delayed expression itself) is known as a suicide proc. When a slot containing a thunk is read, the memory system sends the thunk to a thunk invocation manager, which invokes the suicide proc.

The detailed structure a thunk is presented in Section 3.3.1.

3.2.3 Suicide Procs

Normally, parent and child procedures must communicate to pass arguments and to return results. Delayed expressions, however, are passed their arguments implicitly, and store the result themselves. Thus, the processing of a delayed expression needs no direct linkage with either the process that created it or the processes that are consuming its result. A new procedure call/return mechanism to support this behavior is developed. Procedures that are invoked using this new mechanism are called *suicide procs.*

A suicide proc is invoked by the thunk invocation manager, not by an explicit procedure call. A potential problem, however, is that a thunk generates an independent thread, *i.e.*, a procedure without a parent. This may affect the policy for resource allocation. Currently, running a procedure unfolds into a single execution tree. Once we start spawning processes, there will be an execution forest, and trees will have independent lifetimes. It is likely, however, that the run-time system will deal with this complexity for other reasons. For example, the storage manager and other run-time systems will run independently.

The detailed structure a suicide proc is presented in Section 3.3.2.

3.2.4 The Thunk Invocation Manager

In a dataflow system, a globally defined set of routines known as managers form the run-time system [8]. We need a new manager for spawning suicide procs, the thunk invocation manager, similar to the manager for invoking procedures. In our system, this manager would work as follows.

The thunk invocation manager is passed a thunk. The thunk contains a pointer to a suicide proc, which the thunk invocation manager manager must fetch. The thunk invocation manager must also acquire a new invocation frame for the suicide proc, and send the thunk to the suicide proc. The suicide proc can then read the free variables from the thunk, compute the expression, and store the result back into the delayed slot. Since the suicide proc will deallocate its own invocation frame, no thunk termination manager is needed.

3.3 Compilation Techniques

This section concentrates on the details of compiling lazy data-structures. The detailed structure of a thunk is presented as well as the dataflow graph required to build it. The structure of a suicide proc, the dataflow graph embedding the delayed expression, is also presented.

3.3.1 Thunks

In this section, we make the following conventions:

- `e` is the delayed expression
- `V1, ... , Vn` are the free variables of `e`
- `Se` is the suicide proc that embeds `e`
- `result-address` is a pointer to the delayed slot

A thunk contains a pointer to its suicide proc (`Se`, which embeds delayed expression `e`), the result-address, and the values of all the free variables (`V1, ... , Vn`) of the delayed expression, as shown in Figure 3.8.

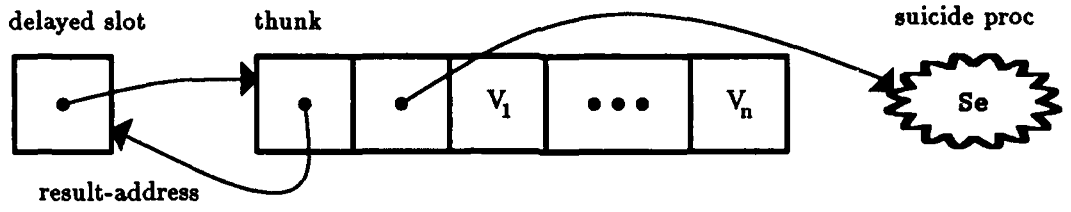


Figure 3.8: A Delayed Slot and its Thunk

At this point we are ready to look at our first dataflow graph. In our figures, every dataflow graph node is labeled with the type of operation executed by the node (in the larger box) and an instruction number (in the smaller box). Immediate constants associated with nodes are written in irregular pentagons above the nodes. Data paths correspond to arcs; inputs correspond to arcs with dangling tails; and outputs correspond to arcs with dangling heads.

The dataflow graph to build a thunk is shown in Figure 3.9. The nodes are numbered z to $z + 2n + 5$; a total of $2n + 6$ operations are required to construct a thunk corresponding to a delayed expression that has n free variables. The numbering does not start at zero to remind us that this dataflow graph is part of a larger dataflow graph.

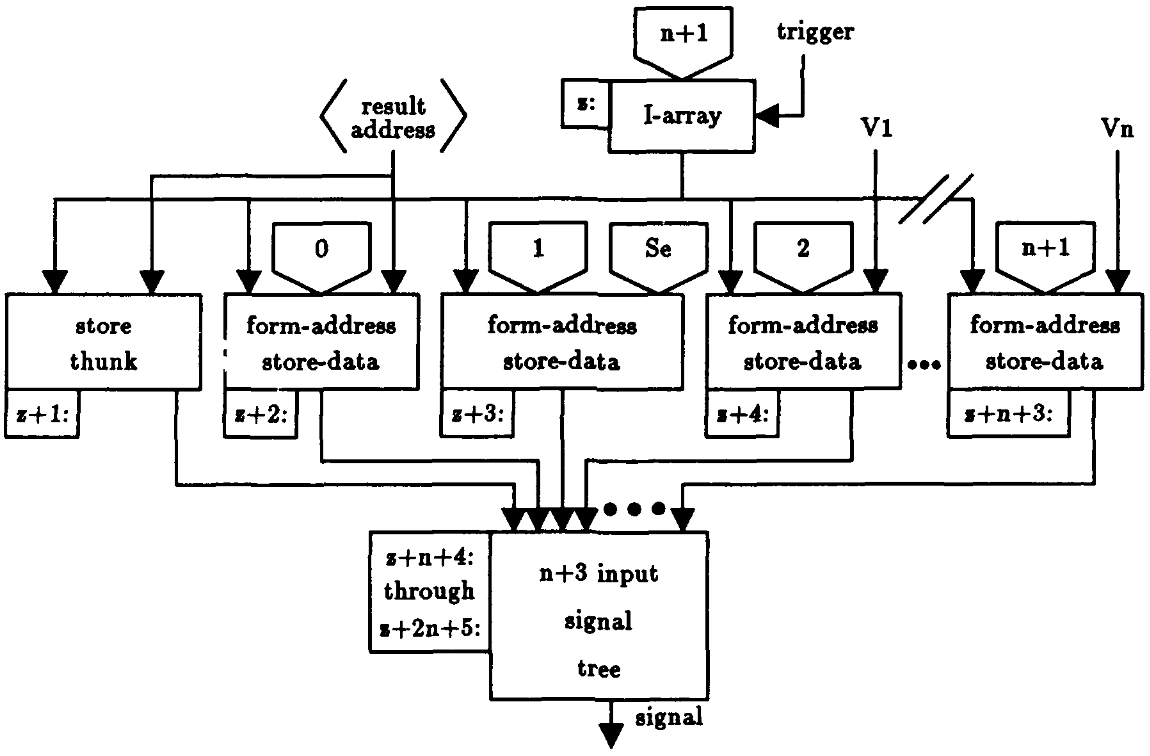


Figure 3.9: The Dataflow Graph to Build a Thunk

Node z (the node labeled z) allocates the thunk as soon as a trigger is arrives. (Almost any token could be used for a trigger. The result-address could be used, for example.) The thunk is sent to node $z+1$ which stores the thunk in the lazy slot (at result-address). The thunk is also sent to nodes $z+2$ through $z+n+3$ which store data at various offsets in the thunk. Node $z+2$ stores the result address in the thunk at offset zero. **`form-address store-data`** is similar to `store-data`, but it accepts accepts structure pointer, offset, and value as inputs instead of an address and a value. An address (like the result-address) can be computed from a structure-pointer and an offset. $z+3$ stores a link to the suicide proc `Se` in the thunk at offset one. The right constant input to node $z+3$ (the link to `Se`) will ultimately get patched at load time to have the proper value. Nodes $z+4$ through $z+n+3$ store the free variables of `e` at offsets two through $n+1$ respectively. All **`store`** nodes produce a termination signal ($n+3$ altogether) which are combined into a single signal by the signal tree. In the TTDA, signal trees are built with binary nodes, so the signal tree in Figure 3.9 would require $(n+3)-1=n+2$ nodes. Monsoon is likely to support higher fan-in for signalling, so fewer nodes would be required.

Note that the thunk is stored at the result-address (node $z+1$) in parallel with the free variables being stored (nodes $z+4$ through $z+n+3$). So if the thunk were spawned (which could not happen before the link to `Se` were in place: node $z+3$ fires) because someone issued a fetch at the result-address (which could not happen before the thunk were in place: node $z+1$ fires), the suicide proc could begin executing before all the free variables are in place. In fact, some free variables may not yet be computed.

The values of the free variables of the delayed expression, the result-address, and a trigger are the inputs to the graph fragment, and the termination signal is the output. A copy of this dataflow graph schema appears each time a slot of a data structure is assigned lazily. Since the number of free variables depends on the expression being delayed, the size of the graph varies.

Some approaches to storage management insist that locatives (addresses of interior structure words) such as the result-address not exist unless a full-fledged pointer to the object exists, and some exclude locatives entirely. Both of these restrictions can be accommodated by expanding the thunk with an additional slot, and storing the pointer to the L-structure and the offset of the delayed slot instead of just storing the result-address.

3.3.2 Suicide Procs

A suicide proc has a single entry point (standard Id procedures have additional entry points which receive arguments). The thunk invocation manager sends the thunk to the suicide procs entry point which is located at a fixed offset in the code for the suicide proc.

The suicide proc unpacks the free variables of the delayed expression as well as the result address. The delayed expression can now proceed normally. When the delayed expression produces a result, the result is stored at the return address, back in the previously delayed slot. After the result is stored and all activity associated with the delayed expression has completed, the suicide proc deallocates its invocation frame and terminates.

The schema for suicide procs is given in Figure 3.10.

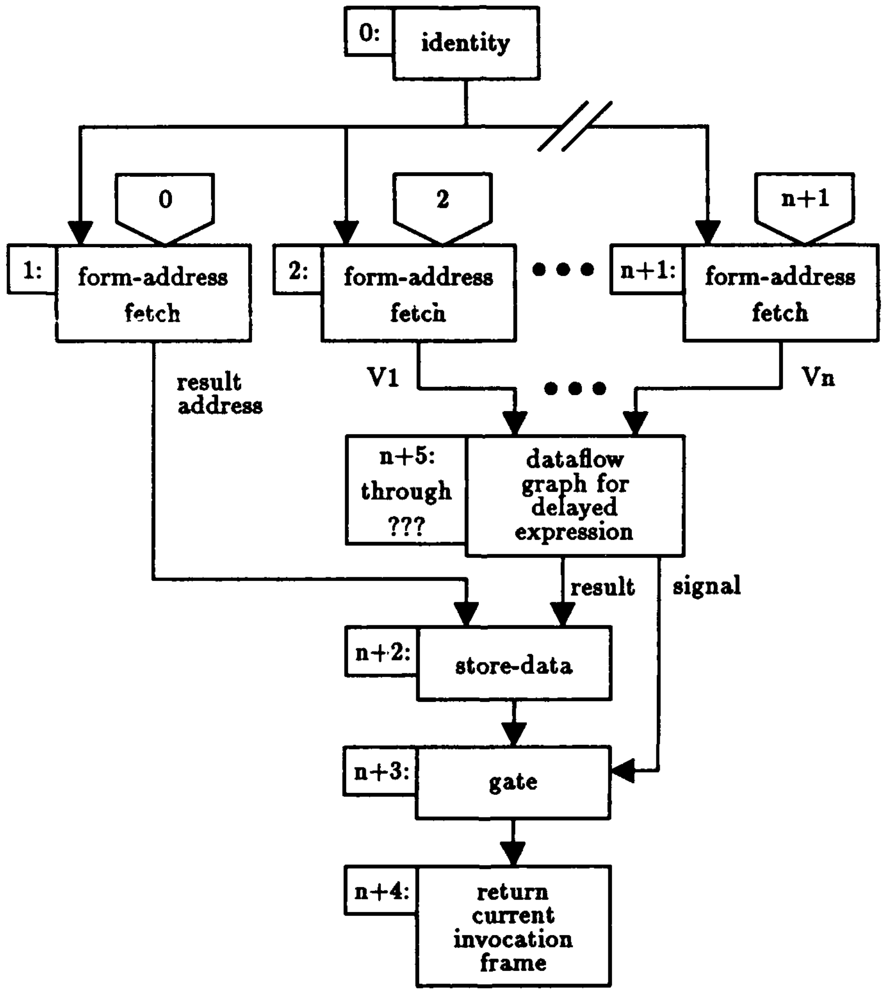


Figure 3.10: Schema for a Suicide Proc

The dataflow graph in Figure 3.10 is numbered from zero to indicate that a suicide proc is not embedded in any larger graph. $n + 5$ nodes are drawn explicitly, and the rest belong to the delayed expression.

Node zero is the entry point for a suicide proc and must ultimately be stored at a fixed offset in the assembled code. Node zero, an **identity**, receives the thunk and distributes it to nodes one through $n+1$. Like the **form-address store-data** nodes in the dataflow graph that loaded the thunk with values (Figure 3.9), the **form-address fetch** nodes retrieve the data stored at the indicated offset in the thunk. Node one fetches the result-address from thunk offset zero. Nodes two through $n + 1$ fetch the values of the free variables V1 through Vn respectively, and deliver them to the graph for the expression that was delayed. The expression graph is embedded directly in the suicide proc (*i.e.*, no additional procedure calls are necessary). The graph for the embedded expression eventually produces a result and (optionally) a termination signal. The result is sent to node $n + 2$, where the return address is also sent. Node $n + 2$ stores the result at the result-address placing a value in the previously delayed slot. Node $n + 2$ produces a termination signal which is combined with the signal generated by the embedded expression (if it generated one) by node $n + 3$ (a gate, which was drawn as a bow-tie in Section 1.5.1). A **gate** node passes on its "top" input when both the "top" and "side" inputs are received. *When node $n + 3$ produces a token, it is guaranteed to be the only remaining token associated* with the suicide prod invocation. Node $n + 4$ consumes this token as it deallocates the current invocation frame.

3.4 Concrete Costs and Benefits

There are two sets of concrete costs to our approach. First are the additional hardware costs, which, as we have seen, are minimal. Two additional states are required for the structure transition diagram. Next is the cost of our thunk mechanism as measured in machine operations. We will consider this cost in greater detail.

In traditional lazy interpreters, we can attribute a cost to each of the following:

1. creating a thunk
2. testing if a thunk has been evaluated
3. evaluating a thunk
4. reading the value

Our hardware support allows for substantial savings. Creating a thunk, evaluating a thunk, and reading a value are done in the usual way. But testing to see if a thunk has been evaluated is better than free; it is implicit. That means that users spend no time or code space in testing or invoking a thunk. The first read of a delayed slot implicitly spawns the corresponding suicide proc, and all successive reads behave in the usual fashion. Conventional systems require repeated checking to see if the thunk has been forced. Furthermore, we can traverse streams exactly like lists, as long as we don't traverse the spine eagerly looking for the end.

The power of our integrated mechanism should not be underestimated. Consider two standard ways of building streams without hardware support. The delayed cell might be explicit, in which case the consumer must make two references each time the stream is walked one step. This technique is pictured abstractly in Figure 3.11, which represents the stream of integers from one, with the first three cell evaluated. The cons cells and the delayed cells alternate. Even if the delayed cell is traversed implicitly, it must be traversed. Concert [22] has hardware support to traverse *futures* (*i.e.*, delayed cells) implicitly, and the garbage collector collapses evaluated futures. Without implicit traversal of the delay (including implicit forcing) a consumer cannot treat this structure as a list.

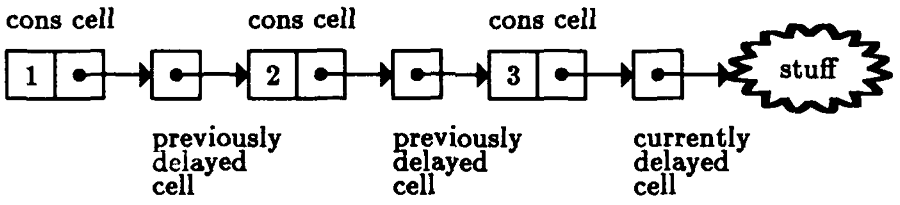


Figure 3.11: A Stream with Explicit Delay Cells

Another possibility is that a flag can be embedded in the stream cell itself. This can be accomplished by expanding the stream cell to a triple. This technique is pictured abstractly in Figure 3.12, which represents the stream of integers from one, with the first three cell evaluated. Unfortunately, we end up with a new the structure is fundamentally different from a list. This technique does not extend easily to other data structures.

In both of these alternatives, primitives for synchronization must be provided, unless the mechanism is implicit.

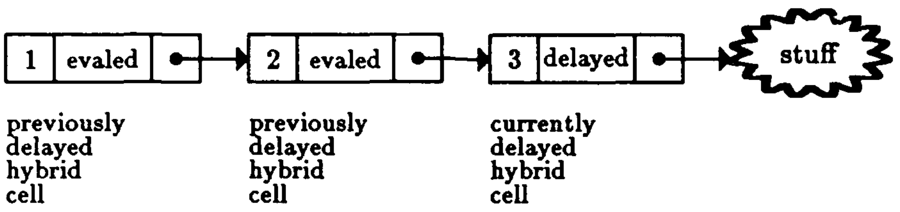

1
evaled
2
evaled
3
delayed
stuff
previously delayed hybrid cell
previously delayed hybrid cell
currently delayed hybrid cell

Figure 3.12: A Stream with "Special Stream-Cons" Cells

3.4.1 In-Thunk Substitution

Id provides a facility for the inline substitution of procedures, allowing programs to use procedural abstraction at no cost, simultaneously trading execution time for program-store space. A programmer may use the **defsubst** keyword in place of the **def** keyword in defining a procedure. Whether or not a procedure is substituted inline or not makes no difference to program behavior; the procedure call version and the inline version have identical semantics. In Id, procedures are declared inlineable at the point of definition. It is also possible to make this decision at the point of call. This possibility is particularly interesting when considered in conjunction with our lazy data-structure mechanism.

A programmer declares a definition inlineable by using the **defsubst** keyword. In this way, a single annotation corresponds to a potentially large number of procedure calls. Note that the **defsubst** annotation is not a directive to inline a procedure, but merely a permission or a suggestion. Consequently, recursive definitions can be marked **defsubst**-able without fear of the compiler's diverging. A recursive **defsubst** cannot be substituted repeatedly until steady state is reached as an infinite chain of ever larger procedures would be generated. Now let us consider the relationship of inlining to thunks.

Consider the process of extending the tail of a recursively defined stream, say the integers, as defined using **ints_from**. Each time the stream is extended, there is a thunk evaluation, as well as a call to **ints_from**. There is a mutual recursion between **ints_from**, a user-defined procedure, and the suicide proc, created by the compiler, but there is only one suspension for each cycle. The natural breaking point is at the suicide proc, for that is where the computation is suspended. By substituting **ints_from** into the suicide proc code block, argument packing and unpacking as well as a manager call to allocate an invocation frame can be avoided. **ints_from** sets up the stream, and from there on the recursion is from suicide proc to suicide proc. Consider the following definitions of **ints** and **ints_from**:

```
typeof ints_from = N -> (list N);
def ints_from n = n:#ints_from (n+1);

typeof ints = list N;
ints = ints_from 1;
```

Each call to `ints_from` builds a thunk. When the thunk is spawned, all it does is call `ints_from` and do some bookkeeping. That call to `ints_from` builds another thunk, with a pointer to the same suicide proc as had the last thunk. All the calls to `ints_from` after the first can be substituted inline.

We are inlining a procedure captured by a suicide proc. Alternatively, we are shifting a procedural recursion broken by a suicide proc, into a suicide proc recursion. We can think of this as unrolling the top half of the first iteration of a loop, and grouping the bottom half of each iteration with the top half of the next.

This shifting of the procedure boundary greatly decreases the marginal cost of delaying expressions. Infinite data-structures must be defined with some sort of recursion, and we can replace the procedural recursion by suicide proc recursion, which is slightly more expensive than procedure recursion, as it must access memory to pass arguments (building and unpacking thunks), but cheaper than both suicide proc recursion and procedure recursion.

3.4.2 Unwinding

The unwinding facility allows the programmer to perform certain kinds of speculative computation. Suspensions are not free, and unwinding allows the programmer to trade suspensions for the possibility of extra work. Consider a system that supports persistent objects. When reconstituting an object (bringing it into the name-space) the entire object need not be reconstituted at once. If the object were a long list, for example, the system might reconstitute as much as fit on one page of memory, and delay reconstituting the rest.

3.4.3 Fetch Elimination

In Id, if a data structure slot were fetched and discarded, the `fetch` would be discarded by the compiler as well as dead code. Why read a value if it won't be used? Now, reading a value can cause a delayed expression to be evaluated. Can we still eliminate lone `fetch`es? After all, it is not possible for a consumer (the reader) to tell if the slot was delayed, in general. If the compiler can prove that the program behavior won't change, then **fetch** elimination is safe. Fetch elimination is not always safe.

In a functional language, it appears that **fetch** elimination is safe. If an expression is discarded, we should not care if it got evaluated since there are no non-local effects. What about I/O? Suppose an array is defined with several lazy slots. A programmer may wish to force the evaluation of several positions before printing the array or before making the array a persistent object. If the fetch elimination problem were not present, each relevant object could be fetched and discarded as we do not want the values, at least for now.

And, of course, non-functional constructs may pose a problem. In Id#, any **fetch** operation might now have a side effect as a result of the presence of I-structures in the language. We alluded to this potential problem when presenting lazy tuples. In the following code, a store to an I-structure is captured in the delayed head of a cons cell:

```
typeof a = I_array N;
a = {a = I_array (1,10);
     a[1] = 1;
     a[2] = 2
     in
       a};

typeof cons_cell = list N;
cons_cell = {a[3] = 3 in 10} #: nil;
```

Whether or not the head of cons_cell is evaluated effects whether or not a[3] is assigned, a non-local effect, perhaps a multiple definition error.

3.4.4 Sequentialization Optimization

When a stream is defined recursively, as it must to be infinite, the size of thunks corresponding to successive elements of the stream form a cycle. The integers, as defined in ints_from, require thunks of uniform size. The following stream, however, requires thunks of two different sizes:

```
typeof needs_2_thunks = list N;
needs_2_thunks = two_free 1 2;

typeof two_free = N -> N -> (list N);
def two_free x y = x :# one_free (x+y);  % thunk has 2 free vars

typeof one_free = N -> (list N);
def one_free y = y :# two_free y (y+1);  % thunk has 1 free var
```

In many cases we can determine this structure statically. Consider the integers, once again. Each thunk contains a pointer to code, a pointer to an L-structure slot, and the value of an index. If we can reuse the thunk, we can avoid allocating and deallocating the thunk itself, and we can avoid rewriting unchanging values. This can be done with some sequentialization, and very little overhead.

Just as a signal tree is used in constructing a thunk (Figure 3.9) to detect when the store operations have completed, additional nodes can usually be added to detect the completion of various events.

Reusing thunk shells is easier than the general problem of rewriting locations in structures since we can guarantee that only one code block points to the thunk at a time. This technique would be restricted to strict expressions, since we must be sure that the last computation was finished with the thunk. Interestingly enough, this reuse of thunks does not interfere with unwinding. A stream can only be suspended in one place, so we never need more than one thunk at a time.

Another possibility is for a stream to retain an invocation frame forever. The feasibility of this depends on the scarcity of invocation frames as well as other engineering issues.

3.4.5 Explicit Deallocation

As we have noted, by the time a suicide proc has read all the thunk slots, it is the only one with a pointer to the thunk. Hence, a suicide proc can always deallocate its thunk explicitly, saving on garbage collection costs.

3.4.6 Unstructured Thunks

Our thunks are flat data-structures. We can take advantage of the particular lexical conditions where the thunk is defined. Sometimes the environments, or parts of the environments can be shared. Furthermore, since each thunk is used exactly once, a tailor made calling convention can be used. Similar ideas were used in the Rabbit [42] and Orbit [29] compilers for specializing procedure calls.

3.5 Summary

This chapter presents the implementation of Id#. L-structures are developed as the implementation mechanism for all lazy data-structures. Then suicide procs (the code embedding delayed expressions), and thunks (the data structures that collects a pointer to a suicide procs and its environment) were presented along with their dataflow graph. The thunk invocation manager, part of the run-time system is described.

Several concrete issues relating to efficiency are also discussed.

Chapter 4

Methodology for Programming with Lazy Data-Structures

Now that we have a mechanism for lazy data-structures, we will exercise it. Because we only allow lazy expressions in certain contexts (the arguments to data constructors) our model is not powerful enough to express all the programs expressible in a lazy functional language. In this chapter we will demonstrate the expressive power and limitations of programming with lazy data-structures.

This chapter is divided into two sections, what we can and can't do with lazy data-structures. We will consider many of the classic programs used by the functional languages community to demonstrate the power of lazy functional languages.

4.1 What We Can Do with Lazy Data-Structures

In this section we show how to code many of the classic examples using lazy data-structures. There are two reasons we have a chance of succeeding in this task, even though our language has less expressive power than a lazy functional language. First, although non-strictness as embodied in Id is weaker than laziness, quite often it will suffice. And second, laziness is usually associated with data structures.

We consider streams, other lazy data-structures, and search programs.

4.1.1 Streams

Streams are the most famous use of laziness[1]. Streams are potentially infinite lists that expand on demand, and provide a uniform interface for controlling uniform data of unpredictable size. In a lazy language, all lists behave like streams. In our language, however, special provisions must be taken to define infinite objects. Consider the stream of positive integers:

```
typeof ints_from = N -> (list N);
def ints_from n = n:#ints_from (n+1);

typeof ints = list N;
ints = ints_from 1;
%= 1 :# 2 :# 3 :# ...
```

Of course, we could have used the special syntax for arithmetiç sequences. The integers might be consumed as follows. The definition of **add_first** uses pattern matching to destructure arguments. The two dots ("..") indicate that the second clause should only be considered if the pattern in the first clause fails to match. Normally, all clauses can be considered in parallel. Only one ".." clause is allowed, and is only considered if the patterns in all other clauses fail to match.

```
typeof add_first_n = N -> (list N) -> N;
def add_first_n 0  s    = 0
 |..add_first_n n (x:xs) = x + add_first_n (n-1) xs;

typeof triangle = N -> N;
def triangle n = add_first_n n ints;
```

As compared to the equivalent program written in a lazy functional language, **ints_from**, the producer, requires a single annotation, and **add_first_n**, the consumer, needs no annotation. Functions that consume streams are identical to the functions that consume lists. The following functions are standard stream producers, transformers, and consumers. Where appropriate, a stream comprehension version is also given (indicated by a trailing underscore).

```
%%% map a unary function over a stream
typeof smap1 = (*0 -> *1) -> (list *0) -> (list *1);
def smap1 f  nil   = nil
 |  smap1 f (x:xs) = f x :# smap1 f xs;
```

[1]We will not engage in a discussion of the utility of stream-style programming, but we will use it as a vehicle for discussing our system.

```
def smap1_ f s = {:# f x || x <- s};
%= f s1 :# f s2 :# s3 :# ...

%%% the squares of the integers
typeof squares = list N;
squares = smap1 {fun x = x*x} (upfrom 1);
squares_ = {:# x*x || x <- upfrom 1};
%= 1 :# 4 :# 9 :# ...

%%% scale a stream by a constant factor
typeof scale = N -> (list N) -> (list N);
def scale a s = smap1 ((*)a) s;
def scale_ a s = {:# a*x || x <- s};
%= a*s1 :# a*s2 :# a*s3 :# ...

%%% the integers based on recursive stream mapping
typeof ints1 = list N;
ints1 = {def suc x = x+1;
         ints = 1 :# smap1 suc ints
         in ints};
```

There is no comprehension for the recurrence ints1. Consider the following function for numerical integration:

```
%%% integrate a function using the rectangle rule
typeof integrate = (N -> N) -> N -> N -> N -> N;
def integrate f lo hi dx =
  { xs = smap1 ((+) lo) (scale dx (upfrom 0));
    ys = smap1 f xs;
    areas = smap1 ((*)dx) ys;
    n = fix ((hi-lo)/dx);
   in
     add_first_n n areas};

typeof integrate_ = (N -> N) -> N -> N -> N -> N;
def integrate_ f lo hi dx =
  { xs = {:# lo+(dx*x) || x <- upfrom 0};
    ys = {:# f x || x <- xs};
    areas = {:# dx*y || y <- ys};
    n = fix ((hi-lo)/dx);
   in
     add_first_n n areas};
```

It is interesting to remove the intermediate streams from these definitions.

```
typeof integrate_1_liner = (N -> N) -> N -> N -> N -> N;
```

```
def integrate_1_liner f lo hi dx =
  { areas = smap1 ((*) dx)
                  (smap1 f
                         (smap1 ((+) lo)
                                (scale dx (upfrom 0))));
   n = fix ((hi-lo)/dx);
  in
    add_first_n n areas};

typeof integrate_1_liner_ = (N -> N) -> N -> N -> N -> N;
def integrate_1_liner_ f lo hi dx =
  { areas = {:# dx * f(lo+(dx*x)) || x <- upfrom 0};
   n = fix ((hi-lo)/dx);
  in
    add_first_n n areas};
```

The mapping version gets cumbersome quickly, but the comprehension version remains compact. We still need to reduce the stream using a separate mechanism. We would like the compiler to have the ability to remove intermediate lists and streams and compile **integrate_** as efficiently as **integrate_1_liner_**, but we will not deal with that topic in this thesis. We continue developing our stream library.

```
%%% map a binary function over two streams
typeof smap2 = (*0 -> *1 -> *2) -> (list *0) -> (list *1) -> (list *2);
def smap2 f (x:xs) (y:ys) = f x y :# smap2 f xs ys
 |..smap2 f   x      y    = nil;
def smap2_ f xs ys = {:# f x y || (x,y) <- lazy_zip2 xs ys};
```

zip2, which is part of the standard environment, turns a pair of lists into a list of pairs. We must define library functions to do lazy zipping.

```
%%% zip two streams lazily
def lazy_zip2 (x:xs) (y:ys) = (x,y) :# lazy_zip2 xs ys
 |..lazy_zip2   x      y    = nil;
```

If we used **zip2** instead of **lazy_zip2** in **smap2**, although the result would be generated lazily the intermediate list of **x**'s and **y**'s would be generated eagerly.

```
%%% first order recurrence given a seed and a recurrence relation
typeof sgen1 = (*0 -> *0) -> *0 -> (list *0);
def sgen1 f x0 = x0 :# sgen1 f (f x0);

%%% the integers as a first order recurrence
```

```
typeof ints2 = list N;
ints2 = sgen1 ((+)1) 1;

%%% second order recurrence given two seeds and a recurrence relation
typeof sgen2 = (*0 -> *0 -> *0) -> *0 -> *0 -> (list *0);
def sgen2 f x0 x1 = x0 :# sgen2 f x1 (f x0 x1);

%% the Fibonacci numbers as a second order recurrence
typeof fibs = N -> N -> (list N);
def fibs a b = sgen2 (+) a b;

%%% filter a stream by a predicate
typeof sfilter = (*0 -> B) -> (list *0) -> (list *0);
def sfilter p  nil    = nil
 | sfilter p (x:xs) = if p x then x :# sfilter p xs
                      else          sfilter p xs;
def sfilter_ p s = {:# x || x <- s when p x};


%%% prefix computation (partial products) on a stream
typeof sexpand = (*0 -> *0 -> *0) -> *0 -> (list *0) -> (list *0);
def sexpand f z  nil    = nil
 | sexpand f z (x:xs) = { val = f z x
                          in
                            val :# sexpand f val xs};

%%% stream of all triangle numbers
typeof triangles = list N;
def triangles = sexpand (+) 0 ints;

%%% incremental dot product of two sequences
typeof dot = (list N) -> (list N) -> (list N);
def dot s1 s2 = sexpand (+) 0 (smap2 (*) s1 s2);
```

Streams have eager heads and lazy tails, and we can implement this behavior precisely in our model. It is difficult and often impossible for a compiler to make such optimizations automatically (in this case to recognize that the heads can be evaluated eagerly). This problem is handled explicitly in our system, *i.e.*, not automatically, but under programmer control.

It is interesting to note which programs have a comprehension syntax. Comprehension handles simple generation and transforming, but not recurrence, compression, or expansion (scanning).

A problem elegantly solved by streams is Hamming's problem [27], generating the ordered list of integers $\mathcal{H}$ containing exclusively 2's, 3's, and 5's as factors.

$$\mathcal{H} = \left\{2^i 3^j 5^k\right\}$$

The following recursive fact leads to the clever solution below:

$$\mathcal{H} = \{1\} \cup 2\mathcal{H} \cup 3\mathcal{H} \cup 5\mathcal{H}$$

```
typeof merge = (list N) -> (list N) -> (list N);
def merge (x:xs) (y:ys) =   % merge two ordered streams; no dups.
  if x<y then      x :# merge xs (y:ys)
  else if x>y then y :# merge (x:xs) ys
       else        x :# merge xs ys;

typeof hamming = list N;
hamming =
  { h235 = 1 : merge (scale 2 h235) h35 ;
    h35  = 3 : merge (scale 3 h35 ) h5  ;
    h5   = 5 :        scale 5 h5
   in
     h235};
```

This is our first example where conses with various degrees of laziness have been mixed and matched. **merge** depends on laziness to break an infinite recursion, but **hamming** can safely call the normal cons function, a more efficient call. If we replaced the ":" symbols in **hamming** with ":#" symbols, the program would behave identically, although slightly less efficiently.

If we switched the stream-conses (":") in **merge** with the list-conses (":#") in **hamming**, the program would diverge. How did we know when to use stream-cons and when to use list-cons? The conses in **hamming** are only executed once, while the conses in **merge** break a recursion.

No collection of stream programs would be complete without a sieve of Eratosthenes for generating primes.

```
%%% THE SIEVE OF ERATOSTHENES

typeof remove_multiples = N -> (list N) -> (list N);
def remove_multiples e s = sfilter {fun x = 0<>(rem x e)} s;

typeof sieve = (list N) -> (list N);
def sieve (x:xs) = x :# sieve (remove_multiples x xs);

typeof primes = list N;
primes = sieve (upfrom 2);
```

Or we can do it another way.

```
%%% THE PRIMES by FILTERING THE INTEGERS
%%% Use the prime stream being generated to filter [3 5 7 ...]

typeof prime_test_given = (list N) -> N -> B;
def prime_test_given (p:ps) x =
  if (p*p) > x then true           % tested to root x; prime
  else if 0 == (rem x p) then false % a composite
       else prime_test_given ps x;

typeof primes1 = list N;
primes1 =
  {primes = 2 :# sfilter (prime_test_given primes) (upfrom 3 by 2)
   in primes};
```

The prime test is strictly a consumer, ignorant of the fact that delaying may be going on. The programs for generating the prime streams each require a single **#**.

A lazy system does not distinguish between lists and streams. We, on the other hand, consume streams and lists similarly, but must produce them differently. By requiring annotations, separate libraries (containing programs such as **map**) for lists and streams must be used.

4.1.2 Other Lazy Data-Structures

In this section we consider two other data structures that can be constructed lazily, trees and memoization tables.

Trees

We define a tree as a leaf or a node with a value and two sub-trees. We also define a function which lazily maps a unary operator over the elements of a tree , a function to add up the numbers in the top few levels of a tree of numbers, and a sample infinite tree of numbers.

```
type tree *0 = leaf | node *0 (tree *0) (tree *0);

typeof map_tree = (*0 -> *0) -> (tree *0) -> (tree *0);
def map_tree f leaf = leaf
 |  map_tree f (node x t1 t2) =
  node (f x) (#map_tree f t1) (#map_tree f t2);

typeof add_n_levels = N -> (tree N) -> N;
def add_n_levels n leaf = 0
```

```
 |  add_n_levels 0 (node x t1 t2) = 0
 |..add_n_levels n (node x t1 t2) =
   x + add_n_levels (n-1) t1 + add_n_levels (n-1) t2;

typeof funny_tree = tree N;
funny_tree =
  {t = node 1 (map_tree ((+)1) t) (map_tree {fun x = x*x} t)
    in
      t};
```

The tree constructor **map_tree** requires a single annotation to delay the expansion of each lazy child. But **add_n_levels**, a tree traversing procedure, requires none.

This scenario covers a wide variety of "AI search problems" such as mini-max and game searching, where the item stored at the node describes a situation reachable from the parent in one step. While the search tree may be finite, it may be enormous (for example, the number of reachable chess positions), but only a small portion of it need be expanded at any time. Searching large spaces fits into the "infinite structures" programming paradigm.

Memoized Functions

Suppose we have a function over a small discrete domain that is expensive to compute, and suppose that we are not likely to need the value of the function for all values of the domain, but we are likely to need some values several times. In such a situation If it may be worthwhile to perform a small amount of work for each element of the domain in order to save computing a few values in the range. The following routine memoizes any function over the specified integer range. Notice that the result is a function.

```
typeof memoize = (N -> *0) -> (N,N) -> (N -> *0);
def memoize f (lo,hi) =
  { memo_array = {array (lo,hi)
                   | [i] # f i || i <- lo to hi};
    def memo_f i = memo_array[i];
    in
      memo_f};
```

This example, clearly in the "expensive slots" programming paradigm, allows the programmer to trade some implicit bookkeeping for potential efficiency. "Why", one might ask, "doesn't the programmer simply manage the bookkeeping explicitly? Just keep a table of flags indicating whether or not the slot has been evaluated, and check that first before computing." No problem, so far. But what happens when you need to change the value of a flag? That facility is not available in a functional language. The technique demonstrated in this section allows the programmer to associate the needed information with each slot, but no more.

4.1.3 Search

We have already mentioned a class of search problems which can be viewed as explicit tree walks. Now we consider a couple of specific search problems.

Eight Queens

The eight queens problem [47] can be solved as a search problem. The problem is to place eight queens on a chess board so that no queen can capture another queen in a single move. Algorithms to generate the ninety-two solutions are well known. The idea is to build solutions one column at a time. A partial solution is an eight row by i column board ($i \leq 8$). Start with an empty zero column solution, and extend all solutions with $i-1$ columns to i columns. Partial solutions with i columns are represented as partial permutations (lists of length i) of the integers from zero to seven. Each integer indicates the corresponding queen's row.

To generate all solutions, the program below uses non-strict data-structures. Laziness is not necessary. `L!i` selects the `ith` element from a list `L` (zero origin is used: `L!0` is the first element of list L).

The `checks` procedure determines if adding a queen in row `q` will endanger the queen already in column `i` of partial-board `board`.

```
typeof checks = N -> (list N) -> N -> B;
def checks q board i =
  { board_i = board!i
   in
     q == board_i or abs (q-board_i) == i+1};
```

The `safe` procedure determines if it is safe to add a queen in row `q` to partial-board `board`. `safe` calls `checks` on each column of `board`.

```
typecf safe = N -> (list N) -> B;
def safe q board = foldr_list (and) true
                            {: not (checks q board i)
                               || i <- 0 to (length board)-1};
```

The queens procedure sets up the partial solution table boards, with a single empty solution with zero columns. All solutions with i-1 columns are extended to i columns for all safe extensions.

```
typeof queens = N -> (list (list N));
def queens n =
  { boards = {array (0,n)
              | [0] = nil:nil
              | [i] = extend i || i <- 1 to n};
    def extend i = {: q:board || board <- boards[i-1]
                                 & q <- 0 to 7
                                 when safe q board}
   in
    boards[n]};
```

The reason we do not need laziness is that the search space is explored in its entirety. Generating the first ten solutions, however, is not as simple, since we do not wish to exhaust the search space. Nonetheless, we can find the solutions by replacing the list enumeration in queens by stream enumeration. Then, only the desired number of solutions will be produced.

```
typeof lazy_queens = N -> (list (list N));
def lazy_queens n =
  { boards = {array (0,n)
              | [0] = nil:nil
              | [i] = extend i || i <- 1 to n};
    def extend i = {:# q:board || board <- boards[i-1]
                                  & q <- 0 to 7
                                  when safe q board}
   in
     boards[n]};
```

The Paraffins

Turner popularized the paraffins problem to demonstrate the power of lazy evaluation and higher-order programming. Since we will refer to this problem several times in the remainder of the thesis, this section is devoted to defining the problem.

A paraffin is a molecule with structural formula C_nH_{2n+2}. Paraffins, also known as the alkanes, are acyclic and have no "double bonds". Methane (CH_4) and iso-butane (C_4H_{10}) are drawn in Figure 4.1.

A paraffin is isomorphic to an acyclic undirected graph, with the carbon atoms mapping to internal nodes, and the hydrogen atoms mapping to leaf nodes. All internal nodes have degree

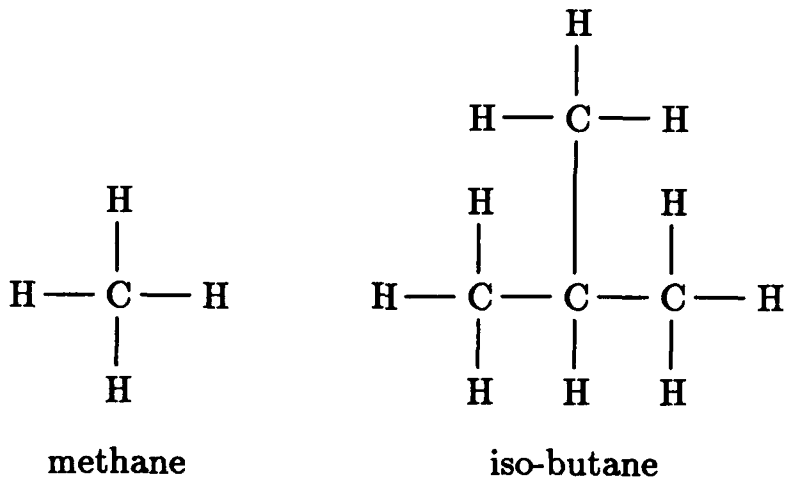


Figure 4.1: Two Paraffins

four, and all leaf nodes have degree one. The graphs corresponding to methane and iso-butane would look exactly like the molecules in 4.1.

Without losing any information, we can ignore the hydrogen atoms and the corresponding leaf nodes, and map a paraffin to an acyclic undirected graph of bounded-degree four. Hydrogen atoms (leaf nodes) are implied to bring the valence (degree) of each carbon atom (internal node) to four. This common simplification makes the pictures less cluttered and is sometimes convenient when discussing some aspects of the problem.

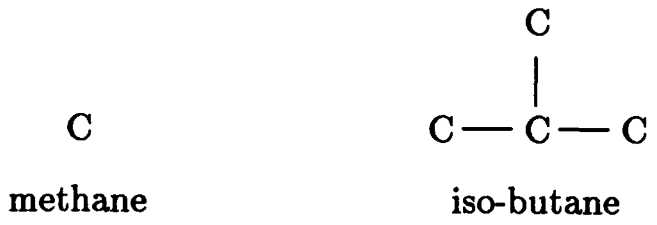


Figure 4.2: Two Simplified Paraffins

If a graph isomorphism exists between the graphs corresponding to two molecules, the molecules are said to be equivalent. If no graph isomorphism exists between the graphs corresponding to two molecules with the same number of carbon atoms, the molecules are said to be isomers (structurally different). In Figure 4.3, all three molecules have the same structural formula (C_6H_{14}). However, (a) and (b) are equivalent, and (c) is an isomer of (a) and (b).

A sub-problem of the paraffins problem involves paraffin radicals (radicals, for short), sub-molecules of paraffins. A radical is a molecule with structural formula C_nH_{2n+1}, a paraffin with one hydrogen atom missing. Alternatively, any bond in a paraffin can be broken to produce

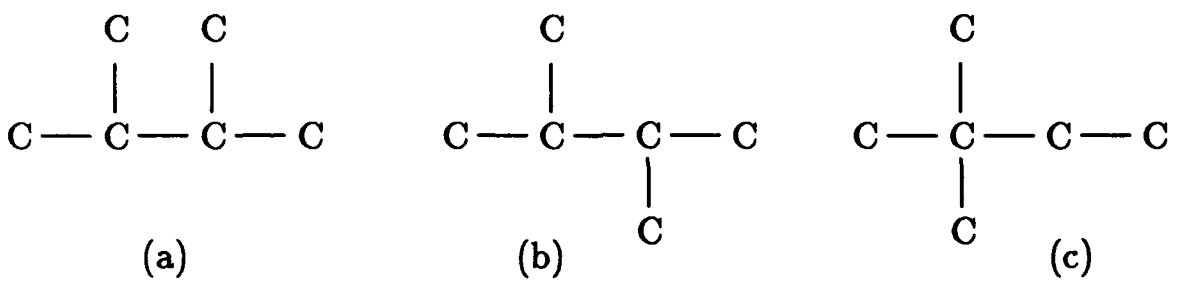


Figure 4.3: Some Isomers of C_6H_{14}

two radicals.

A radical is isomorphic to an unordered ternary tree. The hydrogen atoms correspond to the leaves of the tree. Both the root and the internal nodes have three children, and children are unordered. Two radicals are equivalent if their corresponding trees have an isomorphism. methyl (CH_3) and Ethyl (C_2H_5) are drawn in Figure 4.4.

methyl
ethyl

Figure 4.4: Two Radicals

We will refer to the *size* of a radical or paraffin as the number of carbon atoms contained in it.

The problem is to enumerate, without repetition, the paraffins up to a certain size. The answer should list the paraffins of size one, the paraffins of size two, and so on, up to paraffins of some specified size. "Without repetition" is the tricky part. Just as it is tricky to twist two paraffins around to see if they line up (as we saw in Figure 4.3), a program must not output two equivalent paraffins.

This defines the abstract problem. Before we get into solutions, however, we give an indication of the complexity. To do this, we must get into issue of representation. Radicals can be represented by ternary trees trees. `radical` is defined as a new algebraic type as follows. Although a hydrogen is not normally considered a radical (radical of size zero?: C_0H_1), it is convenient for us to do so.

```
type radical = H | Rad radical radical radical;
```

Representing paraffins, however, is not as easy. Radicals have a root from which to start, but paraffins have no such distinguished nodes. We can simply pick a node, and then hang four radicals from that node as follows:

```
    type paraffin = Para radical radical radical radical;
```

But if we pick a node randomly, there are many representations for a given paraffin. We have even more choices than that, as the children are unordered. So, we see that even our representation for radicals is not unique, as the sub-radicals are unordered, too. The molecule pictured in (a) and (b) of Figure 4.3 has 108 representations! Seventy-two representations correspond to choosing an "outer carbon" as the starting point (or paraffin-origin), and thirty-six representations correspond to choosing an "inner carbon" as the paraffin-origin. The number of representations of paraffins and radicals is exponential in the size.

Now, we have a handle on the problem definition and complexity. We go on to consider two solutions.

Turner's Paraffins Program

Turner's algorithm generates radicals recursively by complete induction[2]. A radical of size n has three smaller sub-radicals whose size adds up to $n - 1$. By making the restriction that the size of the first sub-radical is less than or equal to the size of the second sub-radical, whose size is less than or equal to the size of the third sub-radical, many of the duplicate representations can be avoided for radicals[3]. This restriction also means that the size of the first sub-radical can not be more than a third the total sizes of the sub-radicals, and a corresponding fact for the second sub-radical. Although this implied restriction does not change the number of representations, it makes the generation more efficient. An array of radicals ranging in size from zero to n is generated as follows:

```
    typeof rad_array = N -> (array (list radical));
    def rad_array n =
      { rads = { array (0,n)
               | [0] = H:nil
               | [i] = radgen i || i <- 1 to n};
        def radgen n = {: Rad a b c ||
                            i <- 0 to floor((n-1)/3)
                          & j <- i to floor((n-1-i)/2)
                          & a <- rads[i]
                          & b <- rads[j]
```

[2]By complete induction we mean that the objects of size n depend on objects of sizes less than n.
[3]The first duplicate occurs at size seven; the proof is left as an exercise to the reader.

```
                & c <- rads[n-1-i-j]}
      in
        rads};
```

The rad_array procedure has two bindings. rads, a memoization array for lists of radicals of each size, is generated with an array comprehension. radgen, which depends on the existence of radicals of sizes zero to $i-1$ to generate the radicals of size i, generates radicals using a list comprehension that has generators five levels deep. a, b, and c are the sub-radicals. i, j, and n-1-i-j are the corresponding sizes of the sub-radicals. i ranges over all possible sizes for the first sub-radical, and a ranges over all radicals of size i. Similarly for j and b. c ranges over all radicals of whatever carbon atoms remains. It should be clear that all radicals are generated, although not uniquely.

That's the easy part of the algorithm.

Our representations are not unique. Non-unique representations makes it difficult to tell if two radicals or paraffins are equivalent. And if we ever generate the same paraffin more than once, we have to discard all but one.

When we introduced the radicals, we noted that we can get a radical by starting with a paraffin and "breaking off" a hydrogen atom. Similarly, we can generate a paraffins of size n by starting with a radical of size n and "completing it" by adding a hydrogen atom. Turner's algorithm actually adds a methyl group (CH_3) to a radical that is one size smaller than the size of the desired paraffin.

A single paraffin can be generated by attaching a methyl group to to several distinct radicals. This has nothing to do with representation. The paraffin in (a) and (b) of Figure 4.3 can only be generated in one way by this method. Chopping off any methyl group leaves an identical paraffin. The paraffin in (c) of Figure 4.3 can be generated in two ways. We can chop off a methyl group from a long or short "chain" and leave two different radicals. It should be clear that if we have all radicals of size $i-1$, we can generate all paraffins of size i by attaching methyl groups to the radicals.

Now we have a problem. We can generate all paraffins of a given size, but we will have repetitions. And since the representations are not unique, it is non-trivial to filter out duplicates.

Turner deals with the non-uniqueness issue is by defining equivalence classes for paraffin representations. Two representations of paraffins are in the same equivalence class if the paraffins are equivalent.

The three procedures (or *laws*) **rotate**, **swap**, and **invert** take one representation for a paraffin and return another representation for the same paraffin. **rotate** rotates the paraffin's radicals (the radicals at the top level). **swap** exchanges the paraffin's first two radicals. **invert** shifts the paraffin-origin to the root of the paraffin's first radical. The laws use pattern matching to destructure their arguments.

```
typeof laws = list (paraffin -> paraffin);
laws = rotate:swap:invert:nil;

typeof rotate = paraffin -> paraffin;
def rotate (Para a b c d) = Para b c d a;

typeof swap = paraffin -> paraffin;
def swap   (Para a b c d) = Para b a c d;

typeof invert = paraffin -> paraffin;
def invert (Para H b c d) = Para H b c d
 |  invert (Para (Rad x y z) b c d) = Para x y z (Rad b c d);
```

If applied in the correct order, these three laws can take any representation of a given paraffin into any any other representation for the same paraffin. The closure of these laws over a singleton set containing a paraffin is the set of all representations, *i.e.*, the equivalence class. This may or may not be obvious.

The following procedures generate the equivalence class of a paraffin by taking the closure under the laws. "++" is the list append operator.

```
typeof equivclass = paraffin -> (list paraffin);
def equivclass p = closure_under_laws laws (p:nil);

typeof closure_under_laws = (list (paraffin -> paraffin)) ->
                            (list paraffin) -> (list paraffin);
def closure_under_laws laws s = s ++ closure1 laws s s;

typeof closure1 =
  (list (paraffin -> paraffin)) ->
  (list paraffin) -> (list paraffin) -> (list paraffin);
def closure1 laws s t =
  closure2 laws s (mkset {: p || law <- laws
                              & p   <- map_list law t
                              unless member? (==) p s});

typeof closure2 =
  (list (paraffin -> paraffin)) ->
  (list paraffin) -> (list paraffin) -> (list paraffin);
def closure2 laws s nil = nil
 |..closure2 laws s  t  = t ++ closure1 laws (s++t) t;
```

The mkset procedure (used in closure2 above) removes duplicates from a list.

```
typeof mkset = (list paraffin) -> (list paraffin);
def mkset  nil   = nil
 |  mkset (x:xs) = x : {: y || y <- xs unless x==y};
```

Using equivclass, testing paraffin equivalence is straightforward. The member library routine takes an equality test, an element, and a list, and determines if the element is in the list.

```
typeof equiv = paraffin -> paraffin -> B;
def equiv a b = member? (==) b (equivclass a);
```

Given the ability to test for equivalence, the quotient procedure takes an equivalence relation and a list, and returns a list with no two equivalent elements.

```
typeof quotient = (paraffin->paraffin->B)->
 (list paraffin) -> (list paraffin);
def quotient f  nil   = nil
 |  quotient f (a:x) = a : {: b || b <- quotient f x unless f a b};
```

Finally, we can generate the radicals of size $n-1$, slap on methyl groups, and filter out equivalent representations.

```
typeof paragen = N -> (list paraffin);
def paragen n =
  { radicals = rad_array (n-1);
    rads     = radicals [n-1]
   in
     quotient equiv {: Para r H H H || r <- rads}};
```

Turner's solution uses higher order procedures and abstraction in a big way. In case you didn't notice, the Id# programs in this section have no "#"s in them, *i.e.*, it is not necessary to use lazy data-structures. The fact that closure_under_laws can be executed eagerly in a lazy functional language with no penalty is not obvious, though (perhaps it is?). This fact is very difficult to deduce automatically. We discuss this issue in Section 5.2. The solution does make heavy use of non-strict data-structures to expose parallelism, however.

If the problem were modified slightly, then laziness would come in handy. If the problem were, for example, to produce ten paraffins of a particular size, we wouldn't like to produce all the paraffins of that size, or even all the radicals of smaller sizes. But, how much laziness is required? It turns out, that we need simply convert the list enumerations in **radgen** (in **rad_array**) and in **paragen** to stream enumeration, and we have a "lazy version". It is still true that executing **closure_under_laws** eagerly causes no penalty, but it is even less obvious and no easier to deduce automatically. We discuss this issue in Section 5.2.

Turner's solution is exponentially inefficient. By changing the laws a little bit, and taking advantage of the fact that canonical forms exist, a solution of the same style (equivalence class, closure, etc.) can be developed. For certain canonical forms, it turns out that even more efficient solutions exist, solutions whose running time is linear in the size of the output [6].

An Efficient Paraffins Program

This section presents an efficient solution (linear in the size of the output, which makes it asmytotically optimal) for the paraffins problem. A more detailed description of this solution can be found in [6]. This solution uses no higher order functions or laziness.

The first step is to establish a total order on radicals. This is fairly easy (since radicals have a reference point, the root), and leads directly to a canonical form for radicals. A radical is in canonical form if its sub-radicals are in ascending order, and if the sub-radicals are in canonical form. We will not give the details of the total ordering and canonical form here. Suffice it to say that the **gen_radicals** procedure below generates radicals completely and uniquely. The **tails** procedure produces the list of all prefixes of a list. The **radical** type is redefined to memoize the size.

```
def tails  nil   = nil
 |  tails (a:as) = (a:as) : tails as;

type radical = H | Rad N radical radical radical;

def gen_radicals w =
  { def rgen wp1 =
      {: Rad wp1 r1 r2 r3 ||
        w = wp1-1
        & w1      <- 0 to fix(w/3)
        & r1:r1tl <- tails radicals[w1]
          & w2      <- w1 to fix((w-w1)/2)
          & r2:r2tl <- tails (if w1<w2 then radicals[w2] else r1:r1tl)
            & w3  = w-w1-w2
            & r3 <- if w2<w3 then radicals[w3] else r2:r2tl};
   radicals = { array (0,w)
                | [0] = H : nil
                | [i] = rgen i || i<-1 to w}
   in radicals};
```

Now we need a canonical form for paraffins. The graphs that model paraffins are called *free trees.* Knuth (Volume I) discusses the enumeration (*i.e.*, unique enumeration) of free trees [28]. The trick is to realize that free trees (paraffins) have a well defined *centroid*, which is either one node or a pair of adjacent nodes. The centroid is like the center of mass. We define two types of paraffins, BCPs (bond centered paraffins) and CCPs (carbon centered paraffins) corresponding to the two cases.

```
type paraffin = BCP radical radical |
                CCP radical radical radical radical;
```

BCPs and CCPs partition the paraffins. Canonical form for paraffins is achieved if the radicals of BCPs and CCPs are kept in order and in canonical form.

The paragen procedure generates the paraffins completely and uniquely. bcpgen generates the bond centered paraffins, and ccpgen generates the carbon centered paraffins.

```
def paragen radicals n =
  { def bcpgen w = if (0 <> remainder w 2) then nil
                   else {: BCP r1 r2 ||
                             r1:r1tl <- tails radicals[fix(w/2)]
                           & r2 <- r1:r1tl};
    def ccpgen w =
      {: CCP r1 r2 r3 r4 ||
        w1       <- 0 to fix((w-1)/4)
        & r1:r1tl <- tails radicals[w1]
          & w2      <- w1 to fix((w-1-w1)/3)
          & r2:r2tl <- tails (if w1<w2 then radicals[w2] else r1:r1tl)
            & w3min    = (fix (w/2))-w1-w2
            & w3lo     = max w2 w3min
            & w3      <- w3lo to fix((w-1-w1-w2)/2)
            & r3:r3tl <- tails (if w2<w3 then radicals[w3] else r2:r2tl)
              & w4  = w-1-w1-w2-w3
              & r4 <- if w3<w4 then radicals[w4]
                      else r3:r3tl}
   in bcpgen n, ccpgen n};
```

This solution does not require any laziness or non-strictness. In fact, it can be translated into Fortran in a straightforward way. Extracting parallelism from the Fortran, however, would be very difficult. Consider the **gen_radicals** procedure, which enumerates the radicals of sizes 0 to n. While the procedure *can* be executed in well defined phases, strict semantics would severely limit the available parallelism. (We explore this in Section 5.1.) Id's non-strict semantics, however, harnesses all the parallelism available. This is an excellent example of where laziness provides non-strictness, but all that is needed is the less restrictive and cheaper non-strictness.

If we took up the modification to Turner's original problem statement as presented at the end of the last section, it is clear that we can get a "lazy version" by converting the list enumerations of **rgen**, **bcpgen**, and **ccpgen** to stream enumeration, and we can generate paraffins one at a time.

4.2 What We Cannot Do with Lazy Data-Structures

As we mentioned, our system is not as expressive as a lazy functional language. In this section we explore programs that are difficult or impossible to code using lazy data-structures.

As pointed out by Henderson [24], laziness cannot always be restricted to data structures if we wish to achieve lazy semantics. The following program relies on laziness in an argument to a procedure:

```
typeof problem = N -> (list N);
def problem n = 1 to n ++ problem (n+1);
```

The right argument to append ("++") must be delayed to break an infinite recursion. It is straightforward, however, to produce a program that functions correctly in our restricted system.

```
typeof no_problem = N -> (list N);
def no_problem n =
  { def lazy_range lo hi =
      if lo == hi then hi:nil
      else lo :# lazy_range (lo+1) hi;
    def no_problem_ n = lazy_range 1 n :# no_problem_ (n+1);
    def flatten1 (nil:rest) = flatten1 rest
     | flatten1 ((a:as):rest) = a :# flatten1 (as:rest);
    in
      flatten1 (no_problem_ n)};
```

Suppose we call **no_problem** with an input of 1. Thunks will be associated with:

1. each element of each intermediate list generated by **lazy_range**,
2. each intermediate list as a whole (":#" in **no_problem**),
3. and each element of the result (":#" in **flatten_1**).

Further suppose that the result contains x elements. We will generate $2x + \sqrt{x}$ thunks. A compiler that did a perfect job at analyzing **problem** would generate code that would generate the same number of thunks.

The theoretical minimum is one thunk per output element, as we may suspend any time. We can achieve this in our language by removing abstraction.

```
typeof no_abstraction = N -> (list N);
def no_abstraction n =
  { def no_abstraction_ lo hi =
      { lo1,hi1 = if lo == hi then 1, (hi+1) else (lo+1),hi
       in
         lo :# no_abstraction_ lo1 hi1}
   in
     no_abstraction_ 1 n};
```

If we are willing to allow some unwinding, *i.e.*, some speculation: generating several elements at a time, we can avoid even more thunks. We can generate each intermediate list eagerly, and end up with $\sqrt{x}$ thunks for an output stream of length x.

```
typeof speculation = N -> (list N);
def speculation n =
  { def speculation_ lo hi =
      if lo == hi then hi :# speculation_ 1 (hi+1)
      else lo : speculation_ (lo+1) hi;
    in
      speculation_ 1 n};
```

In the first solution, although there is a lot of abstraction, we coded all the routines (including the library append function) down to primitives. In the latter solutions, we avoided abstraction, and solved the problem more efficiently. Abstraction is interfering with our lazy data-structuring mechanism. Abstraction is the hallmark of functional languages, and any limitation on our ability to abstract is serious.

4.2.1 Lay

A real program with this aforementioned difficulty can be found in the Miranda Standard Environment. The `lay` procedure, when applied to a list of strings, joins them together after appending a newline character to each string. The result is a string. Since Miranda defines strings as lists (at some deep level), strings can be partially defined. The desired behavior is to avoid computing the result string fully unless the entire string is needed. Consider the translation of the Miranda version into Id:

```
typeof lay = (list S) -> S;
def lay  nil   = nil
 | lay (a:x) = string_concat a (string_concat "\n" (lay x));
```

The desired lazy behavior can be achieved by rewriting the library routine `string_concat`. This is a problem, since we don't want the user altering library functions.

4.2.2 Equal Fringe

The problem of comparing the fringes of two trees for equality is a famous demonstration of the power of laziness with respect to non-infinite data-structures. The following code lazily converts the trees' fringes into streams, and then compares the stream elements until a difference is encountered.

The trees being compared may be lazily constructed or not, but at least part of the fringe must be reachable in finite time.

```
type tree = Leaf N | Node tree tree;

typeof equal_fringe = tree -> tree -> B;
def equal_fringe t1 t2 = equal_list (fringe t1) (fringe t2);

typeof equal_list = (list N) -> (list N) -> B;
def equal_list nil    nil    = true
 |  equal_list (a:as) nil    = false
 |  equal_list nil    (b:bs) = false
 |  equal_list (a:as) (b:bs) = if a==b then equal_list as bs
                               else false;
```

We still need to generate the fringe. The following program, which works in a lazy functional language, relies on the lazy evaluation of a procedure argument, as did Henderson's example. Furthermore, it cannot be trivially modified to run correctly in our system.

```
typeof fringe = tree -> (list N);
def fringe t = fringe_ t nil;

typeof fringe_ = tree -> (list N);
def fringe_ (Leaf x) tail = x:tail
 | fringe_ (Node t1 t2) tail = fringe_ t1 (fringe_ t2 tail);
```

The problem is that we need to delay the second argument to the `fringe_` procedure. Our trick of changing "`:`" to "`:#`" won't help us here.

Fortunately, there are other ways to fringe a tree. Here are two. The first shifts the tree to the right, preserving the fringe, until the left edge of the fringe is near the top.

```
typeof fringe1 = tree -> (list N);
def fringe1 (Leaf x) = x:nil
 | fringe1 (Node t1 t2) = fringe1_ t1 t2;

typeof fringe1_ = tree -> tree -> (list N);
def fringe1_ (Leaf x) t = x :# fringe1 t
 | fringe1_ (Node t1 t2) t3 = fringe1_ t1 (Node t2 t3);
```

Or we can simply keep track of the spine along which we descended.

```
typeof fringe2 = tree -> (list N);
def fringe2 t = fringe2_ t nil;

typeof fringe2_ = tree -> (list tree) -> (list N);
def fringe2_ (Leaf x) nil    = x:nil
 | fringe2_ (Leaf x) (t:ts) = x :# fringe2_ t ts
 | fringe2_ (Node t1 t2) ts = fringe2_ t1 (t2:ts);
```

It is interesting to compare the different solutions. The first recursive call to `fringe_` needs a delayed argument, but the second recursive call does not. If a compiler only compiles one version of a procedure, no amount of strictness analysis will optimize this away. The laziest (and most expensive) case must be applied universally.

Continuing with the `fringe_` procedure, since the traversal requires one procedure call for each node in the graph, there is a delayed and subsequently forced expression for both the internal and fringe nodes. So, on the average, there are two delays and two forces for each fringe element generated.

Now consider either of the other solutions. There is exactly one force and one delay per fringe element generated, the theoretical minimum.

The point is not investigating clever algorithms *per se.* The original algorithm is slick, but surprisingly expensive. The other algorithms delay only what is necessary.

4.2.3 The Non-Strict Tree

The following example is drawn from [26] where it is credited to Bird [12]. In a single traversal, it replaces each element of the fringe by the minimum element of the original fringe. Although this program is featured as depending on lazy evaluation, it only requires non-strict evaluation [35]. A new variety of tree is used that has values at the leaves but not at the internal nodes.

```
type ntree = tip N | fork ntree ntree;

typeof traverse = ntree -> N -> (ntree,N);
def traverse (tip n)   x = (tip x), n
 | traverse (fork L R) x = { L1,xL = traverse L x;
                             R1,xR = traverse R x;
                             x1 = min xL xR
                            In
                              (fork L1 R1), x1} ;

typeof solution_2 = ntree -> ntree;
def solution_2 t = { t1,x = traverse t x
                    In
                      t1} ;
```

It is interesting to ask if we can produce the the result lazily in our language, and the answer is: not very easily. Again, the problem is abstraction over data-structure construction. But, no piece of the answer can be produced before the entire input is traversed. So, if the result is produced lazily, we are effectively doing two passes anyway, and much more straightforward algorithms are possible.

4.3 Power and Limitations of Lazy Data-Structures

We have considered a host of examples to gain experience with the power and limitations of our system. We conclude with a discussion of the power and limitations of lazy data-structures.

4.3.1 More Expressive Power

We can now write programs with streams and other lazy data-structures, an ability not previously available in Id. Lazy arrays are a particularly dramatic example, as they provide a power in excess of that available in a lazy functional language. This is somewhat slippery, as arrays are not typically available in functional languages. Unless array generation is extremely restricted, it is not possible to propagate demands through an array, as index functions are not always available or invertible. Consider the following lazy array comprehension:

```
{ array (1,n) | [perm i] # f i || i <- 1 to n}
```

In order to compute a slot value, the producer must be determined. But this cannot be done without at least some evaluation of the index function. In general, we will have to evaluate the index function until it generates the desired value. There is no other way to "propagate demand" unless we can invert the index function.

In our language, no pretense is made about laziness with respect to the index calculation: the index calculations are performed eagerly, determining the producer of each slot, but the value expressions are left unevaluated. Work is done to put a thunk in every lazy slot, and, in that way, the producer of the value for each slot is manifest. Of course, some slots can be filled eagerly, some lazily, and some not at all.

4.3.2 Lack of Fidelity

What you see is what you get. What you meant, on the other hand, may be somewhat different. For example, we must be careful to make a distinction between the following two expressions. The first one forces the potentially delayed tail of **s** and then includes it in a delayed expression, perhaps too late to delay. The second one does not risk any premature forcing.

```
{(h:t) = s in h :# t}

{(h:_) = s in h} :# {(_:t)= s in t}
```

The second form, which is more conservative, is clumsy. This subtlety arises in practice. Consider the following function to filter a stream:

```
typeof sfilter = (*0 -> B) -> (list *0) -> (list *0);
def sfilter p  nil    = nil
 |  sfilter p (x:xs) = if p x then x :# sfilter p xs
                        else          sfilter p xs;
```

Even though **xs** is named in both branches of the conditional, the consequent is a stream cons and might never be evaluated. But, as defined, **xs** always gets a value. A more conservative definition would be the following:

```
def sfilter p nil = nil
 |..sfilter p  s  =
  { (x:_) = s
   in
     if p x then x :# sfilter p {_:xs = s in xs}
     else             sfilter p {_:xs = s in xs}};
```

So, our original definition looked fine, but it did a little more work. This is not a problem, normally, where we are concerned with controlling infinite streams, but it does extra computation, and might not be safe.

The following program has a similar difficulty and demonstrates the difficulty precisely. We would like a program that touches the first n elements of a stream, but the following, contrary to our intuition, touches n+1 elements:

```
typeof take = N -> (list *0) -> (list *0);
def take 0   s    = nil
 |..take n (x:xs) = x : take (n-1) xs;
```

It has been suggested that this problem is due to the way we compile pattern matching [36]. We could move the selection of slots "inward" to the lexical use, so that they are not selected in as many cases. This idea seems right and merits additional study.

4.3.3 Limitations of the Lazy Data-Structures

Our system is not as expressive as a lazy system. There are two ways that we see the difference. First, arguments to procedures are not delayed. And second, we cannot abstract over lazy constructors and preserve laziness. The second point, while subsumed by the first, is independently interesting.

We provide no mechanism for abstracting over lazy data-structures. This is analogous to the problem present in eager languages of abstracting over conditionals. Consider the following user-defined conditional (equivalent to Lisp's **and** function, which is also sequential):

```
typeof if_nil = (list *0) -> (list *1) -> (list *1);
def if_nil lst val = if nil == lst then nil else val;
```

According to eager semantics, the actual expression for val would be evaluated before if_nil were applied. All applications (under an eager interpreter) of the following function for making a copy of a list would run off the end of the list and produce an error. A lazy system would interpret the program as desired.

```
    typeof copy_list = (list *0) -> (list *0);
    def copy_list lst = if_nil lst {(a:as) = lst in a : copy_lst as};
```

This inability to abstract over conditionals is analogous to our inability to abstract over lazy data-constructors. By the time an expression reaches a lazy data-constructor, it is already evaluated.

4.4 Summary

This chapter demonstrates the utility of our approach as well as the limitations through a collection of examples. We are able to deal with stream programming to a large extent, and notice that new library routines are needed. As we tackle more complex problems, this deficiency emerges as a difficulty with abstracting over lazy assignment in general.

If we can solve a problem in Id#, we end up with fine grain control over what is lazy and what is not. Many of the classical examples can be solved in Id#. We hypothesize that this is due primarily to the following two reasons. Laziness is often used to achieve non-strict behavior. Our system already has non-strict behavior, obviating this use for laziness. Also, laziness is typically associated with data structures.

Chapter 5

Conclusions

We have presented an extension to the the dataflow language Id for supporting lazy data-structures, as well as extensions to the compiler and architecture. In this, the concluding chapter, we present a demonstration of the cost of data structure strictness, a challenge for the advocates of lazy functional languages, and concluding discussions.

5.1 A Demonstration: the Cost of Data Structure Strictness

Although conventional languages are strict in procedure arguments, data structures may be passed around before they are completely defined, *i.e.*, non-strictly, and synchronization between producer and consumer is done explicitly. In a "conventional multiprocessor", both for efficiency and manageability, synchronization is likely performed on a large grain basis. For example, a vector may be made readable only after it is completely defined. Or, a matrix may be made available one row at a time. Individual synchronization can be viewed as enforcing a strictness constraint equivalent to data availability. And, blocked synchronization is analogous to building entire data-structures strictly; all the data must be in place before any values are made available.

Data structure strictness, however, is not free. We demonstrate the cost of the explicit synchronization of data-structures with some examples.

5.1.1 Paraffins

Consider the algorithm of Section 4.1.3 for efficiently enumerating the paraffins (molecules with structural formula C_nH_{2n+2}). The radicals (C_nH_{2n+1}) are defined using complete induction. As a group, the radicals of size n (n carbon atoms) depend on the radicals of sizes 0 to $n-1$.

The following Id code allows the radicals to be computed with or without barriers between the production of radicals of successive sizes. A barrier insures that the preceding phase completes before the succeeding phase begins.

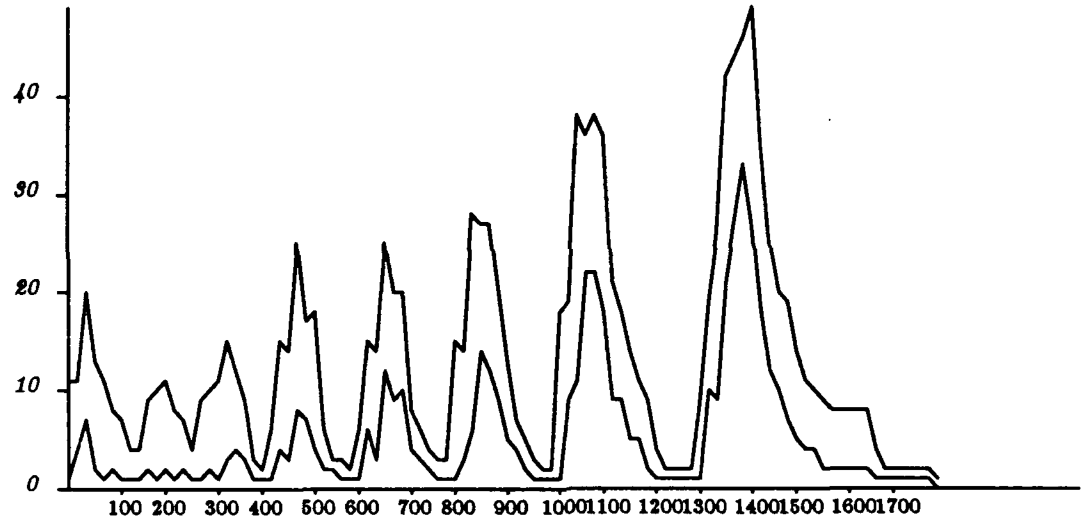
```
type radical = H | Rad N radical radical radical;

typeof traverse_radical = radical -> N;
def traverse_radical H = 0
 |..traverse_radical (RAD n r1 r2 r3) =
   { tr1 = traverse_radical r1;
     tr2 = traverse_radical r2;
     tr3 = traverse_radical r3
    in
      if (strict n) then (tr1 + tr2 + tr3) else 0};

typeof traverse_radical_list = (list radical) -> N;
def traverse_radical_list  nil  = 0
 |..traverse_radical_list (r:x) =
   1 + (traverse_radical r) + (traverse_radical_list x);

typeof traverse_radical_lists = (array (list radical)) -> (array N);
def traverse_radical_lists radicals =
  { (lo,hi) = bounds radicals
   in
     { array (lo,hi)
      | [i] = traverse_radical_list radicals[i]
              || i <- lo to hi}};
```

traverse_radical_lists, which traverses each list of radicals and returns an array of the number of radicals of every size, implements the barrier. All operations associated with any of the traverse procedures, however, are masked from the collected statistics.

The **gen_rads_bar** procedure defines the array, **strict_array**; the *i*th slot becomes defined when all the radicals of size i are fully defined. If the extra argument **barrier?** to **rad_gen_bar** is **true**, radicals of size $n-1$ are completely defined before any radicals of size n are derived.

```
typeof gen_rads_bar = N -> B -> (array N);
def gen_rads_bar w barrier? =
  { def rgen wp1 gate1 =
      {: Rad wp1 r1 r2 r3 ||
         w = wp1-1
       & w1      <- fix(0*gate1) to fix(w/3)
       & r1:r1tl <- tails radicals[w1]
       & w2      <- w1 to fix((w-w1)/2)
       & r2:r2tl <- tails (if w1<w2 then radicals[w2] else r1:r1tl)
      & w3  = w-w1-w2
      & r3 <- if w2<w3 then radicals[w3] else r2:r2tl};
   radicals =
     { array (0,w)
      | [0] = H : nil
      | [i] = rgen i strict_array[if barrier? then i-1 else 0]
              || i<-1 to w};
   strict_array = traverse_radical_lists radicals
  in strict_array};
```

The parallelism profile[1] in Figure 5.1 describes the construction of the radicals of size eight or less, where the construction of any radicals of size n does not begin until all radicals of size $n-1$ are completed. Barriers are present between the construction of radicals of different sizes. The domain is time steps, and the range is the number of parallel ALU operations.



ALU OPERATIONS PROFILE IN GEN'RADS'BAR (8 $T)

Figure 5.1: Parallelism Profile for Strict Generation of Radicals

The parallelism profile in Figure 5.2 describes the construction of the radicals of size eight or less, where the construction of radicals of all sizes begins as soon as possible in an overlapped and non-strict fashion.

In the strict case, 15,805 operations are performed, and the critical path is 1788. In the non-strict case, 15,805 operations are performed, and the critical path is 705. In the original algorithm, 15,708 operations are performed, and the critical path is 681. The extra operations are the result of the procedure linkage to **traverse_radical_lists**, and the conditional barrier.

[1]A parallelism profile plots the number of operations that can be executed in parallel for a particular program and input under a greedy schedule. The two curves envelope the actual locus.

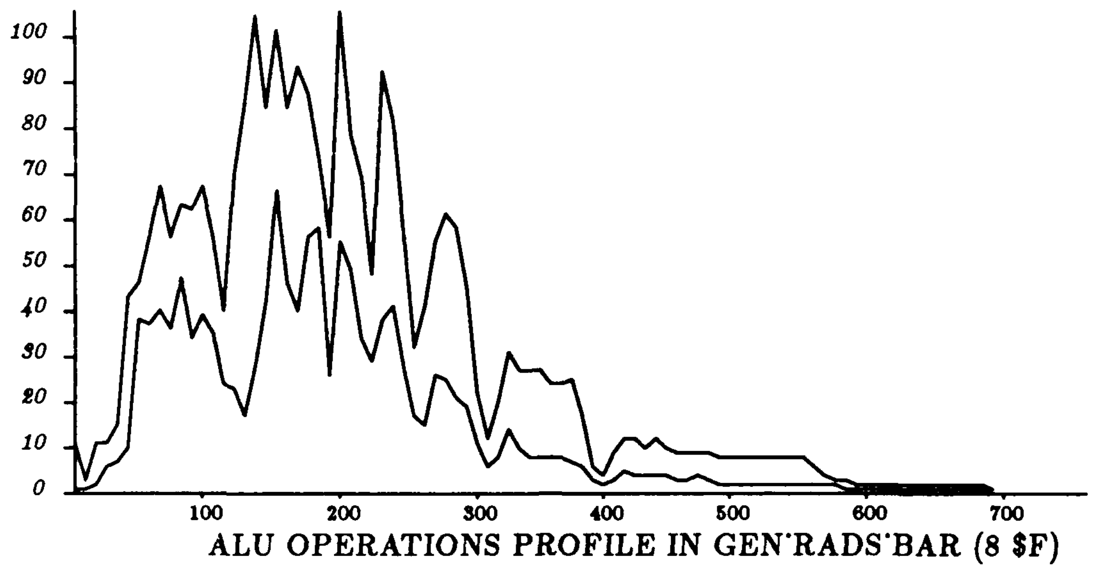


Figure 5.2: Parallelism Profile for Non-Strict Generation of Radicals

The actual operations associated with the traversal procedures are masked from the statistics.

Strictness implies serialization and costs us parallelism. The non-strict case has a shorter critical path and more parallelism. The shape of the parallelism profile is important, too. The many constrictions make it difficult to keep a parallel machine busy. Also, synchronization must be performed explicitly; the masked operations associated with **traverse_radical_lists** accounted for the majority of the operations.

5.1.2 Insertion Sort

In this section we consider a functional implementation of insertion sort. A list is sorted by successively inserting each element into a sorted list. **insert_elements** calls **insert_element** to insert each element of the original list into a sorted intermediate result. **insertion_sort** calls **insert_elements** with the initial list, and an empty list (which is vacuously sorted).

```
typeof insertion_sort = (list n) -> (list n);
def insertion_sort as = insert_elements as nil;

typeof insert_elements = (list n) -> (list n) -> (list n);
def insert_elements  nil   sorted_list = sorted_list
 | insert_elements (a:as) sorted_list =
     insert_elements as (insert_element a sorted_list);

typeof insert_element = n -> (list n) -> (list n);
def insert_element a  nil   = a:nil
```

```
|  insert_element a (x:xs) =
     if a < x then
       a:x:xs
     else
       x:insert_element a xs;
```

If we enforce data-structure strictness, insertions proceed one element at a time. The next insertion cannot begin until the previous one has completed. The synchronization, however, is simpler than in the paraffins example, as the list is traversed sequentially. So, when we reach the correct position, the prefix must be defined. Or must it? Doesn't this inductive fact simplify synchronization? Not necessarily. Unless we enforce extraneous synchronization along the way, the "heads" of the prefix cells could be proceeding in parallel. So, we must synchronize the entire list.

A non-strict approach allows the successive elements to be inserted in a pipelined and overlapped fashion. As soon as a prefix is computed, it is returned, partially defined, and the next element can begin its crawl down the list.

What if we destructively update a shared list that has some method for locking? Although Id has no facilities for this destructive behavior, we hypothesize the results for a parallel system that allows updates such as Halstead's Multilisp on Concert [21, 22]. The same mechanism that is used for locking may subsume our producer consumer synchronization. After all, that is what locking and exclusive access are all about.

5.1.3 Discussion

Two points merit exposition:

- Strictness limits parallelism, as it implies sequentialization.
- Synchronization is expensive. Fine grain synchronization of data can be accomplished using I-structure memory. Large grain synchronization, *which implies the coordination of a set of fine grain synchronizations*, is more complex in an unordered environment. In a SIMD machine this is easy as computation proceeds in lock step, but the synchronization cost is paid constantly. In a MIMD machine, where flexible evaluation order buys utilization, coordinated synchronization is expensive.

5.2 A Challenge: The Difficulty of Optimization

Strictness analysis is a compilation technique for deducing information about argument strictness. If a procedure is strict in an argument, that argument need not be passed lazily. Abstract interpretation [32, 16, 25] and context analysis [46] are two techniques for strictness analysis. Since the problem is unsolvable in general, these approaches, as well as any other approaches to strictness analysis, are necessarily approximation techniques.

Consider the `closure_under_laws` procedure of Section 4.1.3. It produces the equivalence class of a particular paraffin, *i.e.*, all the legal representations. There is no need for laziness in this procedure, but this is not easy to show. After a particular paraffin, say P_0, is generated, no equivalent paraffin should be enumerated in the result. Suppose P_1 is next in the enumeration. Before P_1 can be produced we must check that it is not in the equivalence class of P_0, at which point the equivalence class of P_0 must be fully expanded. Before finding P_1, however, we may have checked P_2, P_3, and P_4, only to discover they were equivalent to P_0. Each replica will have expanded the equivalence class of P_0 to include itself. However, at no quiescent point of the algorithm are any equivalence classes partially expanded.

Why is it difficult to deduce the strictness of `closure_under_laws`?

1. The strictness must be deduced from context. It is the way that `closure_under_laws` is called that leads to the strictness.

2. Only some of the calls to `closure_under_laws` require the result to be fully defined. It happens that the algorithm loops until it makes one of these "strict calls".

Strictness analysis in the presence of higher-order functions and data structures is already complex. In order to produce efficient compiled code for `closure_under_laws`, however, we must have algorithmic insight, a difficult task for a compiler.

5.3 Conclusions

5.3.1 Thunk Efficiency

We have claimed that our mechanism is efficient. In the introductory chapter, we described several key efficiency issues that are important when we implement delayed computation. Have we addressed these issues?

1. sharing

2. the use of thunks where lazy evaluation is not necessary

3. the speed of the operations required to manipulate thunks

1. Sharing is accomplished naturally. An expression is memoized in the data structure in which it belongs, and is computed at most once.

2. Delaying is imposed at the request of the programmer, and, although extra delaying may occur, it is unlikely.

3. The operations are efficient. Creating a thunk is no more than storing the environment, and evaluating a delayed expression is about as complex as applying a procedure. The big win is in subsequent references to the value, which incur no overhead resulting from the fact that the value was once a delayed expression.

5.3.2 Embedding Henderson's System

As we mentioned in the introductory chapter, the framework presented for lazy structures can embed a Henderson-style source-to-source transformation for achieving lazy behavior. Consider the following transformations. `Delay` and `Force` are implemented with lazy data-structures.

```
delay <exp> => {vector (0,0) [0] # <exp>}

force <exp> => exp[0]
```

Although this approach is roughly as efficient as standard solutions, we do not advocate it. Any Henderson-style system is likely to end up with too many thunks and be too expensive.

5.3.3 Variations and Future Directions

We have developed a technique which depends on explicit delaying and implicit forcing. The implicit part relies on a hardware mechanism for synchronization and, not surprisingly, is very efficient. If a similar mechanism were available to trap delayed values so that normal processing were uninterrupted, we might investigate delaying expressions in general. Trapping such delayed expressions can be done, given hardware support in many scenarios, if we are willing to give up sharing. We might view such a system as similar to Lisp's invisible forwarding pointer system.

Several related approaches are worth considering. All of the following possibilities are variations of the language or the semantics but preserve the underlying implementation.

In the cases that data structures are involved, lazy structures can be thought of as simply an optimization in any lazy language on a tagged architecture. Rather than have a thunk resident in a data structure slot, use structure tag-bits to implement L-structures, cutting out the intermediate storage. Implicit forcing comes as a bonus.

Another possibility is eager evaluation for all expressions except those destined for data structure slots. In order to guarantee termination, it may suffice to guarantee that all circular expressions are cut by data-structures, in much the same way that circular combinational logic must be cut by storage elements. Either strictness analysis or annotations may be useful for generating optimized programs. This approach is similar to Burtons approach [15].

Similarly, we can assign even structure slots eagerly, except those that contain pointers to other structures. We could benefit from compiler analysis (type checking) and annotations, both eager and lazy.

The repercussions of these varied choices are not clear. An interesting approach involves using the techniques of Lucassen [31] and Young [48]. Lucassen developed techniques for categorizing expressions as *pure* and *side-effecting*. Due to the presence of the I-structure language construct in Id, changing an expression's evaluation semantics to eager or lazy may change its meaning. Analysis similar to Lucassen's may avoid these difficulties. Young developed techniques for approximating the cost of expressions and approximating termination behavior. If the cost of evaluating an expression is cheaper than the cost of building a thunk, then eager evaluation is more efficient, assuming, of course, that the meaning of the expression is the same.

5.3.4 Concluding Remarks

Our thesis can be stated concisely as follows. An eager non-strict language plus lazy data structures with otherwise eager semantics provides most of the expressive power of a lazy functional language and an opportunity for implementation at close to the cost of an implementation of an eager non-strict language.

The language Id#, Id plus lazy data-structures, was presented in Chapter 2, and its expressive power as well as its limitations were demonstrated in Chapter 4. Chapter 3 presents an efficient implementation.

This much is clear: most expressions can be evaluated eagerly with no loss of expressive power (chance of non-termination). An eager functional language is already very close to the target. A lazy functional language, on the other hand, is far from the target, and, if we can get there at all, it is sure to be a struggle. Great progress has been made in strictness analysis, recently including non-flat domains [20, 46] (*i.e.*, data structures). But, it is unlikely that we will reach the target. We have already pointed out the problem in analyzing indexed structures. Furthermore, the problem of higher-order functions and data structures simultaneously appears quite difficult.

To the purist, who is unwilling to accept any annotation, we simply note that, as demonstrated, these annotations fit naturally into the source language, and have clear semantics. Furthermore, the repercussions are clear, and confined.

For the eager non-strict programmer, who is willing to take responsibility for her program's actions, we hope to have opened up a gamut of possibilities, both powerful and efficient.

Bibliography

[1] J. Adler and L. Drew. The enigma of autistic savants. *Newsweek*, page 54, January 16, 1989.

[2] M. Amamiya. Parallel and pipeline executions in symbol manipulations. Seminar and Personal Communication, November 1987.

[3] M. Amamiya. Data flow computing and parallel reduction machine. *Future Generations Computer Systems*, 4(1):53–67, August 1988.

[4] K. R. Apt. Introduction to logic programming. Technical Report TR-87-35, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, September 1987. Also: Report CS-R8741, Center for Mathematics and Computer Science, P.O.Box 4079, 1009 AB Amsterdam, The Netherlands.

[5] Arvind, K. P. Gostelow, and W. Plouffe. The (preliminary) id report. Technical Report 114, Department of Information and Computer Science, University of California, Irvine, CA, 1978.

[6] Arvind, S. Heller, and R. S. Nikhil. Programming Generality and Parallel Computers. In *Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy*, September 18-22 1988. Also: CSG Memo 287, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.

[7] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing: The dataflow solution. Computation Structures Group Memo 226-3, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, August 1985.

[8] Arvind and R. S. Nikhil. Executing a program on the Massachusetts Institute of Technology tagged-token dataflow architecture. Computation Structures Group Memo 271, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, March 1987.

[9] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structure for parallel computing. In *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer-Verlag, October 1986.

[10] Arvind and R. E. Thomas. I-structures: An efficient data type for parallel machines. Technical Memo TM-178, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, September 1980.

[11] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, New York, 1988.

[12] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(4):239–250, 1984.

[13] A. Bloss and P. Hudak. Path semantics. In *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 476–489. Springer-Verlag, April 1988.

[14] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, September 1988.

[15] F. W. Burton. Functional programming for concurrent and distributed computing. *The Computer Journal*, 30(5):437–450, October 1987.

[16] C. Clack and S. L. Peyton-Jones. Strictness analysis—a practical approach. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 35–49. Springer-Verlag, September 1985.

[17] K. L. Clark and S-A. Tarnlund. *Logic Programming.* Academic Press, New York, 1982.

[18] J. B. Dennis. The origin of the term: "thunk". Personal Communication, May 1987.

[19] M. Hill *et. al.* Design decisions in SPUR. *IEEE Computer*, 19(10):8–22, November 1986.

[20] C. V. Hall and D. S. Wise. Compiling strictness into streams. In *Conference Record of the 14th Annual ACM Symposium on the Principles of Programming Languages*, pages 132–143. Association for Computing Machinery, January 1987.

[21] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Tranactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[22] R. H. Halstead, Jr., T. I. Anderson, R. B. Osborne, and T. I. Sterling. Concert: Design of a multiprocessor development system. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 40–48. IEEE, June 1986.

[23] S. K. Heller. An I-Structure Memory Controller (ISMC). Master's thesis, MIT Deptartment of Electrical Engineering and Computer Science, May 1983.

[24] P. Henderson. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs NJ, 1980.

[25] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Conference Record of the 13th Annual ACM Symposium on the Principles of Programming Languages*, pages 97–109. Association for Computing Machinery, January 1986.

[26] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer-Verlag, September 1987.

[27] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, Amsterdam, August 1977. North-Holland.

[28] D. E. Knuth. *The Art of Computer Programming, Volume 1/ Fundamental Algorithms.* Addison Wesley, Reading, Massachusetts, 1973.

[29] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986. (Proceedings of the SIGPLAN 86 Symposium on Compiler Construction).

[30] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. November 3, 1988.

[31] J. M. Lucassen. *Fluent Languages: A Framework for Combining Functional and Imperative Programming.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, August 1987.

[32] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer-Verlag, April 1980.

[33] R. S. Nikhil. Id world ref rence manual. Computation structures group memo, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, April 1987.

[34] R. S. Nikhil. ID. Computation Structures Group Memo 284, Massachusetts Institute of Technology, March 1988. Version 88.0.

[35] R. S. Nikhil. Lenience and not laziness. Electronic Communication, April 1988.

[36] R. S. Nikhil. Pattern matching in Id. Private Communication, December 1988.

[37] R. S. Nikhil, K. Pingali, and Arvind. Id nouveau. Computation Structures Group Memo 265, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, July 1986.

[38] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, Massachusetts Institute of Technology, Cambridge MA, August 1988.

[39] S. L. Peyton-Jones. FLIC—a functional language intermediate code. Internal Note 2048, University College London Department of Computer Science, London, November 1986.

[40] K. K. Pingali. *Demand-Driven Evaluation on Dataflow Machines.* PhD thesis, Massachusetts Institute of Technology, Cambridge MA, June 1986.

[41] J. Rees and W. Clinger. Revised^3 report on the algorithmic language scheme. Technical report, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambride MA, 1986.

[42] G. L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge MA, May 1978.

[43] K. R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report TR-370, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge MA, August 1986.

[44] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, September 1985.

[45] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Paterson. Architecture of SOAR: Smalltalk on a RISC. In *ACM/IEEE 11th Symposium on Computer Architecture*, pages 188–197, June 1984.

[46] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 386–407. Springer-Verlag, September 1987.

[47] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1976.

[48] J. Young. *The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages.* PhD thesis, Yale University, Department of Computer Science, 1989.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy